Titre: Title:	Traçage de logiciels bénéficiant d'accélération graphique
Auteur: Author:	David Couturier
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Couturier, D. (2015). Traçage de logiciels bénéficiant d'accélération graphique [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1736/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1736/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

TRAÇAGE DE LOGICIELS BÉNÉFICIANT D'ACCÉLÉRATION GRAPHIQUE

DAVID COUTURIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
MAI 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

TRAÇAGE DE LOGICIELS BÉNÉFICIANT D'ACCÉLÉRATION GRAPHIQUE

présenté par : COUTURIER David
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. OZELL Benoit, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BILODEAU Guillaume-Alexandre, Ph. D., membre

REMERCIEMENTS

Pour commencer, j'aimerais remercier mon directeur de recherche pour le support et les conseils judicieux qu'il a su me donner au cours de cette période intense. Ses connaissances techniques dans le domaine surpassent de beaucoup mes attentes et ont aidé à bien déterminer les objectifs de ma maîtrise. Sa disponibilité, sa bonne humeur et son encadrement exemplaire en font un excellent directeur de recherche.

Les membres de l'équipe du laboratoire DORSAL constituent une grande famille : l'entraide et le partage des connaissances ont aussi rendu la tâche beaucoup plus intéressante. Je tiens à remercier spécifiquement Geneviève, Suchakra, Francis et Julien pour leurs explications du fonctionnement de LTTng et de certaines fonctionnalités du noyau système de Linux. Sans leur aide et leurs conseils, l'accomplissement de ma recherche aurait nécessité beaucoup plus de temps.

Merci aussi à Ericsson, EfficiOS et au Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG) pour leur financement et l'achat de matériel spécifique à mon sujet de recherche. Leur participation a rendu possible l'accomplissement de ma maîtrise.

Finalement, j'aimerais remercier ma famille, mais surtout ma soeur avec qui je vis depuis plusieurs mois et qui a démontré une extrême patience et compréhension tout au long de ma maîtrise. Le support exceptionnel de mes parents tout au long de mes études est aussi quelque chose à ne pas oublier !

RÉSUMÉ

En programmation, les récents changements d'architecture comme les processeurs à plusieurs cœurs de calcul rendirent la synchronisation des tâches qui y sont exécuté plus complexe à analyser. Pour y remédier, des outils de traçage comme LTTng furent implémentés dans l'optique de fournir des outils d'analyse de processus tout en gardant en tête les défis qu'implique les systèmes multi-cœur. Une seconde révolution dans le monde de l'informatique, les accélérateurs graphiques, créa alors un autre besoin de traçage. Les manufacturiers d'accélérateurs graphiques fournirent alors des outils d'analyse pour accélérateurs graphiques. Ces derniers permettent d'analyser l'exécution de commandes sur les accélérateurs graphiques.

Ce mémoire apporte une solution au manque d'outil de traçage unifié entre le système hôte (le processeur central (CPU)) et l'exécution de noyaux de calcul OpenCL sur le périphérique (l'accélérateur graphique (GPU)). Par unifié, nous référons à la capacité d'un outil de prise de traces à collecter une trace du noyau de l'hôte sur lequel un périphérique d'accélération graphique est présent en plus de la trace d'exécution du périphérique d'accélération graphique.

L'objectif initial principal de ce mémoire avaient été définis comme suit : fournir un outil de traçage et les méthodes d'analyse qui permettent d'acquérir simultanément les traces de l'accélérateur graphique et du processeur central. En plus de l'objectif principal, les objectifs secondaires ajoutaient des critères de performance et de visualisation des traces enregistrés par la solution que ce mémoire présente. Les différentes notions de recherche explorés ont permis d'établir de hypothèses de départ. Ces dernières mentionnaient que le format de trace *Common Trace Format* (CTF) semblait permettre l'enregistrent de traces à faible surcoût et que des travaux précédents permettront d'effectuer la synchronisation entre les différents espaces temporels du CPU et du GPU.

La solution présentée, *OpenCL User Space Tracepoint* (CLUST) consiste en une librairie qui remplace les symboles de la librairie de calcul GPGPU OpenCL. Pour l'utiliser, elle doit être chargée dynamiquement avant de lancer le programme à tracer. Elle instrumente ensuite toutes les fonctions d'OpenCL grâce aux points de trace LTTng-UST, permettant alors d'enregistrer les appels et de gérer les événements asynchrones communs aux GPUs.

La performance de la librairie faisant partie des objectifs de départ, une analyse de la performance des différents cas d'utilisation de cette dernière démontre son faible surcoût : pour les charges de travail d'une taille raisonnable, un surcoût variant entre 0.5 % et 2 % fut mesuré.

Cet accomplissement ouvre la porte à plusieurs cas d'utilisation. Effectivement, considérant le faible surcoût d'utilisation, CLUST ne représente pas seulement un outil qui permet l'acquisition de traces pour aider au développement de programmes mais peut aussi servir en tant qu'enregistreur permanent dans les systèmes critiques. La fonction "d'enregistreur de vol" de LTTng permet d'enregistrer une trace au disque seulement lorsque requis : l'ajout de données concernant l'état du GPU peut se révéler être un précieux avantage pour diagnostiquer un problème sur un serveur de production. Le tout sans ralentir le système de façon significative.

ABSTRACT

In the world of computing, programmers now have to face the complex challenges that multi-core processors have brought. To address this problem, tracing frameworks such as LTTng were implemented to provide tools to analyze multi-core systems without adding a major overhead on the system. Recently, Graphical Processing Units (GPUs) started a new revolution: General Purpose Graphical Processing Unit (GPGPU) computing. This allows programs to offload their parallel computation sections to the ultra parallel architecture that GPUs offer. Unfortunately, the tracing tools that were provided by the GPU manufacturers did not interoperate with CPU tracing.

We propose a solution, *OpenCL User Space Tracepoint* (CLUST), that enables OpenCL GPGPU computing tracing as an extension to the LTTng kernel tracer. This allows unifying the CPU trace and the GPU trace in one efficient format that enables advanced trace viewing and analysis, to include both models in the analysis and therefore provide more information to the programmer.

The objectives of this thesis are to provide a low overhead unified CPU-GPU tracing extension of LTTng, the required algorithms to perform trace domain synchronization between the CPU and the GPU time source domain, and provide a visualization model for the unified traces. As foundation work, we determined that already existing GPU tracing techniques could incorporate well with LTTng, and that trace synchronization algorithms already presented could be used to synchronize the CPU trace with the GPU trace.

Therefore, we demonstrate the low overhead characteristics of the CLUST tracing library for typical applications under different use cases. The unified CPU-GPU tracing overhead is also measured to be insignificant (less than 2%) for a typical GPGPU application. Moreover, we use synchronization methods to determine the trace domain synchronization value between both traces.

This solution is a more complete and robust implementation that provides the programmer with the required tools, never before implemented, in the hope of helping programmers develop more efficient OpenCL applications.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ACE	Asynchronous Compute Engines
AMD	Advanced Micro Devices
ARM	Advanced RISC Machines
API	Application Programming Interface
APU	Accelerated Processing Unit
BTS	Branch Tree Store
CLUST	OpenCL User Space Tracepoint
CPU	Central Processing Unit
CTF	Common Trace Format
CUDA	Compute Unified Device Architecture
CUPTI	CUDA Profiling Tools Interface
DDR	Double Data Rate
GCN	Graphics Cores Next
GDB	GNU Project Debugger
GDDR	Graphics Double Data Rate
GNU	GNU's Not Unix
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
LTT	Linux Tracing Toolkit
LTTng	Linux Tracing Toolkit next generation
PCI	Peripheral Component Interconnect Express
PCIe	PCI Express
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OS	Operating System
MPI	Message Passing Interface
NOP	No Operation
NTP	Network Time Protocol
RISC	Reduced instruction set computing
STM	System Trace Macrocell
SIMD	Single Instruction Multiple Data
TAU	Tuning and Analysis Utilities
UST	User-Space Tracing

USDT	User Statically Defined Tracing
XML	Extensible Markup Language

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

La complexité des logiciels ne cessant d'augmenter, les programmeurs ont de plus en plus besoin d'utiliser des outils complémentaires afin de pouvoir diagnostiquer les erreurs et problèmes d'exécution de leurs programmes. Effectivement, de nos jours, l'apparition de processeurs à plusieurs cœurs et de périphériques asynchrones comme les accélérateurs graphiques rendent beaucoup plus difficile le diagnostic des problèmes reliés à la synchronisation entre plusieurs fils d'exécution ou même entre processus. L'utilisation d'un débogueur comme GNU's Not Unix *(GNU) Project Debugger* (GDB) n'est plus suffisant étant donné que les problèmes sont souvent dépendants du temps et que ces derniers ne surviennent pas de façon consistante mais plutôt aléatoire.

C'est dans cette optique que le traçage logiciel vu le jour. Une trace système permet entre autres de visualiser les appels systèmes et les changements de contexte des processus et de ces fils d'exécution (si applicable). Des outils d'analyse de trace comme `Trace Compass` permettent de faire l'analyse de ces traces. Des algorithmes y sont constamment développés pour aider le programmeur à faire des liens entre les changements de contexte et les conséquences sur l'exécution du programme. Plusieurs outils de traçage existent et ils présentent tous des différences d'implémentation entre les uns et les autres.

Dernièrement, la grandissante popularité des accélérateurs graphiques pour le calcul ouvrit de toutes nouvelles possibilités aux programmeurs et certains proposèrent des outils spécifiques pour permettre d'analyser l'exécution des programmes sur ces derniers.

Bien que ces outils semblent combler un manque au niveau de l'analyse de l'exécution des programmes utilisant l'accélération graphique, certaines problématiques restent toujours à résoudre.

1.1 Définitions et concepts de base

Afin de comprendre ce travail, certains concepts de base doivent être définis.

1.1.1 CPU vs GPU

Dans ce mémoire, pour signifier la différence entre le *Central Processing Unit* (CPU) et le *Graphical Processing Unit* (GPU), différents termes sont souvent utilisés. En voici quelques-uns qui se réfèrent au GPU : l'**accélérateur matériel**, l'**accélérateur graphique**, le **périphérique**, la **carte graphique** ainsi que le **GPGPU**. Il est important de prendre en

considération que ces termes sont tous des synonymes. En contrepartie l'ordinateur dans lequel se trouve le GPU sera référé comme étant le **CPU**, le **système** ou l'**hôte**. Ces trois derniers termes sont aussi à considérer comme étant synonymes.

1.1.2 Traçage

Le traçage logiciel consiste en l'enregistrement d'informations concernant un ou plusieurs processus. Cette pratique permet d'obtenir des informations pouvant aider les programmeurs à diagnostiquer des problèmes concernant l'exécution de leur programme. Une façon simple d'enregistrer une trace d'un logiciel est d'utiliser la sortie standard (*stdout*) pour y imprimer l'état de notre programme à différents points au cours de l'exécution de ce dernier.

Afin d'encadrer cette pratique et de permettre aux programmeurs d'obtenir des informations complémentaires par rapport à l'exécution des autres programmes s'exécutant simultanément sur le système, des outils de traçage ont étés développés. Ces derniers permettent notamment d'enregistrer des informations qui concernent le noyau du système d'exploitation en plus de fournir des outils aux programmeurs pour tracer de façon efficace certaines portions de leur programme.

1.1.3 Événement & point de trace

Ce mémoire fait souvent référence à des **événements** et des **points de trace**. La différence entre ces deux termes est subtile mais existante : un événement consiste en l'information qui est généralement enregistrée à l'exécution d'un point de trace. Un point de trace est alors défini comme un appel de fonction, dans le programme, qui crée l'événement.

1.1.4 Instrumentation

La pratique d'instrumentation consiste à ajouter des points de trace à l'intérieur du code source d'un programme. Pour ce faire, il est obligatoire d'avoir le code source du programme que nous désirons instrumenter, sans quoi nous devons nous restreindre aux informations qu'une trace du noyau système peut nous fournir pour connaître le comportement d'un programme non-instrumenté. L'instrumentation de programmes doit être faite à l'aide de points de trace au niveau utilisateur, tandis que l'instrumentation du système d'exploitation requiert des points de trace au niveau noyau. Une bonne instrumentation permet d'activer et de désactiver les points de traces à l'exécution (*at runtime*) sans pour autant ralentir le programme instrumenté.

1.2 Éléments de la problématique

Initialement, *Linux Trace Toolkit next generation* (LTTng) était un outil plutôt complet en ce qui concerne la prise de traces sur les processeurs x86 (Toupin, 2011). Malheureusement pour l'outil, la technologie de l'information ne cesse de s'épanouir : LTTng doit donc s'adapter à ces changements d'utilisation des technologies de l'information. L'arrivée d'accélérateurs matériels apporte un nouvel aspect que les outils de traçage existants ne couvrent pas et qui implique une autre dimension de complexité. Il s'agit aussi d'une constatation que Juckeland (2012) tire de l'utilisation d'accélérateurs matériel. Dans un chapitre complet dédié à l'analyse de la performance des accélérateurs matériels, il note que pour bien comprendre le fonctionnement des accélérateurs matériels, trois défis doivent être relevés : la gestion du comportement asynchrone des accélérateurs matériels, la synchronisation temporelle entre l'hôte et le périphérique, et la collecte de métriques de performance spécifiques à l'accélérateur matériel analysé (Juckeland, 2012). G. Jukeland réussit à couvrir l'essentiel de la problématique en lien avec les accélérateurs matériels. Les outils que fournissent les manufacturiers de coprocesseurs graphiques offrent seulement des *Application Programming Interface* (API) de traçage asynchrones où le programmeur peut s'inscrire à des procédures de rappel (dans le cas de la librairie de profilage CUPTI de NVIDIA) (NVIDIA, 2014). Sinon, il peut aussi utiliser des tampons d'événements associés aux fonctions dans OpenCL et en extraire les métriques de temps à l'aide d'opérations définies dans les spécifications de la librairie OpenCL (Group, 2010). Le deuxième point amené par G. Jukeland est le fait que l'horloge du processeur central n'est pas nécessairement synchronisée avec celle de l'accélérateur matériel : les données de temps extraites via une des deux méthodes décrites précédemment doivent donc être synchronisées. Finalement, il mentionne aussi qu'il est important d'accéder à de l'information additionnelle, c'est-à-dire que chaque accélérateur matériel possède des métriques spécifiques à l'application pour laquelle il a été conçu. Dans le cas des accélérateurs graphiques, des données de performance, dont le pourcentage d'utilisation des cœurs de calcul, sont des données importantes à enregistrer et afficher à l'utilisateur pour mieux comprendre la performance de l'application tracée.

1.3 Objectifs de recherche

Il existe déjà des outils qui permettent de tracer les coprocesseurs graphiques. Malheureusement pour ces derniers, ils permettent seulement de tracer l'utilisation du coprocesseur graphique et font abstraction du reste de l'ensemble de l'ordinateur. Avoir une trace du noyau système de l'exécution d'un programme utilisant une accélération graphique (ce que

LTTng permet présentement) n'est pas assez pour complètement comprendre l'ensemble des problèmes d'un programme. L'inverse est aussi vrai : avoir une trace de l'accélérateur graphique qui ne prend pas en compte l'exécution du programme sur l'hôte (le CPU) ne fournit pas un portrait complet.

L'objectif principal de ce travail consiste à fournir un outil de traçage et les méthodes d'analyse qui permettent d'étudier simultanément les traces de l'accélérateur graphique et du processeur central.

Questions de recherche

L'objectif de recherche principal donne naissance à plusieurs questions de recherche. Les principales que nous allons adresser dans ce mémoire sont les suivantes :

Q.1 Est-il possible de récolter des données de traçage en temps réel sans ralentir de façon significative l'exécution du programme analysé ?

Q.2 Est-il possible de faire le lien entre la trace de l'accélérateur graphique et celle de l'hôte tout en respectant les métriques de synchronisation du temps ?

Objectifs spécifiques

Pour arriver à répondre à ces questions de recherche, des objectifs supplémentaires découlant d'une granularisation de l'objectif principal peuvent être déterminés.

O.1 Instrumenter les librairies de calculs pour accélérateurs graphiques.

O.2 Obtenir des mesures de performance et vérifier l'impact du traçage sur le système.

O.3 Implémenter une technique de synchronisation entre la trace de l'hôte et celle de l'accélérateur graphique.

Hypothèses scientifiques

Pour chaque objectif spécifique, nous pouvons émettre des hypothèses auxquelles cette recherche tentera de répondre. À la fin de ce mémoire, nous reviendrons sur ces hypothèses de départ pour vérifier si elles s'avèrent véridiques.

H.1 Il est possible d'intercepter les appels à la librairie OpenCL, d'instrumenter les appels et d'obtenir des données d'exécution par rapport aux commandes graphiques.

H.2 Les techniques de traçage qu'offre la librairie LTTng pour les points de trace en mode utilisateur (LTTng-UST) permettent une collecte rapide des données de traçage sans impact

significatif sur le système. LTTng utilise un tampon circulaire par cœur et enregistre ensuite les données en utilisant le Common Trace Format (CTF) (Desnoyers, 2014). La combinaison de ces deux technologies consiste en une base solide dont les traces de coprocesseur peuvent bénéficier.

H.3 Dû au modèle de communication entre le coprocesseur graphique et l'hôte, il est possible d'utiliser des techniques de synchronisation hors ligne comme celle proposée par Poirier et al. (2010) et d'obtenir une métrique d'ajustement temporelle afin d'être en mesure de synchroniser les traces côte à côte.

Originalité et critères de "succès"

La performance de LTTng ayant été démontrée, l'implémentation proposée devra maintenir le standard. Le fonctionnement des programmes peut, dans des cas rares mais réels, être affecté par le ralentissement dû à la prise de trace. Il est donc primordial de maintenir un impact aussi minime que possible afin d'éviter que la prise de trace influence le comportement du programme tracé.

De plus, le traçage en mode analyse de coprocesseur seulement (sans traçage de l'hôte) devra avoir un faible surcoût. Le critère de succès optimal serait d'être capable d'obtenir une trace mixte de l'hôte et du périphérique sur tout le système.

1.4 Plan du mémoire

Ce mémoire possède six différents chapitres. Le premier chapitre est l'introduction.

Le second consiste en une revue de littérature. L'objectif de cette section est d'informer le lecteur sur les différentes technologies et sujets de recherche qui existent présentement dans le domaine des accélérateurs graphiques et des calculs sur accélérateurs graphiques.

Ensuite, dans le chapitre 3, la méthodologie de recherche est abordée plus en profondeur. Ce chapitre discute de la configuration et des méthodes utilisées pour obtenir des données expérimentales. Le fonctionnement de la solution apportée par cette recherche est aussi expliqué plus en détail.

Puis, le chapitre principal (chapitre 4) aborde la solution entière que présente cette recherche sous forme d'un article scientifique. Ce chapitre discute des différentes plateformes de traçage disponibles pour le CPU et ceux disponibles pour le GPU. L'architecture de la solution est présentée, suivie des détails d'implémentation qui découlent des caractéristiques d'architecture vues précédemment. Finalement, ce chapitre présente les résultats de performance de

la solution, étant donné que les objectifs de recherche incluent des critères de performance à rencontrer.

Pour continuer, une analyse complémentaire est présentée dans le chapitre 5. Le surcoût de la solution proposée combinée avec le traçage noyau de LTTng est mesuré et discuté. Aussi, une discussion générale présente les options d'utilisation de la librairie *OpenCL User Space Tracepoint* (CLUST) en fonction de sa performance. Les contributions connexes à ce travail sont répertoriées à la fin de ce chapitre.

Finalement, le mémoire se complète avec le chapitre 6 : la conclusion. Ce chapitre présente une révision des travaux suivis d'une revue objective sur les limitations de la solution proposée et des améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Traçage logiciel du processeur central

2.1.1 Niveaux d'exécution

Les programmes peuvent s'exécuter à deux niveaux différents. La plupart des programmes (à l'exception de ceux du système d'exploitation) sont écrits pour fonctionner principalement au niveau *utilisateur*. Lorsqu'ils font appel à des fonctions du système d'exploitation, ces appels sont alors exécutés à un autre niveau. Ce deuxième niveau est dit *noyau*. C'est notamment à des fins de sécurité qu'il existe une différence entre ces deux niveaux.

Noyau

Une caractéristique prédominante des outils de traçage est que ces derniers permettent d'analyser et de collecter des traces d'exécution au niveau système et des données de performance sans nécessiter l'instrumentation du code dans le programme à analyser. Ceci est possible étant donné que ces outils sont implémentés pour s'attacher à des sondes dans le noyau du système. Étant donné que les programmes font des appels système, il est possible d'analyser le comportement de ces derniers en observant les appels systèmes effectués.

Utilisateur

Il est aussi possible d'analyser l'exécution des programmes à l'autre niveau : *utilisateur*. Cette technique permet à un programmeur de modifier le code source de son application afin d'y insérer des points de trace et d'analyser le comportement de son application. Cette technique offre un bon moyen aux programmeurs de tracer le fonctionnement d'applications hautement parallèles sans trop perturber leur fonctionnement normal.

2.1.2 Outils Disponibles

Pour les logiciels utilisant principalement les ressources du processeur central, plusieurs options sont disponibles. On compte notamment les outils `strace`, `DTrace`, `SystemTap` et `LTTng`, ainsi que plusieurs autres alternatives.

strace

`Strace` est l'un des outils les plus simples et faciles d'utilisation. C'est dans le début des années 1990 qu'il fut publié. Il est facile de l'utiliser pour obtenir un enregistrement des appels système qu'un processus effectue (strace project, 2010). Pour obtenir une trace `strace`, l'utilisateur doit lancer le programme via `strace` en donnant en paramètre à `strace` le nom de la commande à tracer (ex. "`strace ./monProgramme`"). On peut aussi spécifier des filtres et restreindre la trace à certains appels systèmes, comme l'exemple qui suit où on limite les appels système observés aux appels *open* et *close* de la commande *ls*.

```
1  strace -e open,close ls
2  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
3  close(3)                                     = 0
4  open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
5  close(3)                                     = 0
6  open("/lib/x86_64-linux-gnu/libacl.so.1", O_RDONLY|O_CLOEXEC) = 3
7  close(3)                                     = 0
8  open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
9  close(3)                                     = 0
10 open("/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
11 close(3)                                     = 0
12 open("/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
13 close(3)                                     = 0
14 open("/lib/x86_64-linux-gnu/libattr.so.1", O_RDONLY|O_CLOEXEC) = 3
15 close(3)                                     = 0
16 open("/proc/filesystems", O_RDONLY)          = 3
17 close(3)                                     = 0
18 open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
19 close(3)                                     = 0
20 close(3)                                     = 0
21 Documents
22 close(1)                                     = 0
23 close(2)                                     = 0
24 +++ exited with 0 +++
```

L'exemple ci-dessus illustre bien la facilité qu'offre `strace` pour enregistrer l'ouverture et la fermeture de fichiers de n'importe quel processus. C'est cette facilité d'utilisation qui a fait son succès.

Malheureusement, `strace` n'offre pas la meilleure performance face aux processus qui utilisent plusieurs fils d'exécution. Cette faiblesse est discutée dans un article qui propose un format d'enregistrement sur canaux multiples pour limiter l'utilisation de primitives de synchronisation entre les fils d'exécution (Spear et al., 2012).

Par défaut, `strace` permet de tracer un processus, mais il peut aussi tracer différent processus en spécifiant leur identifiant de processus (*pid*).

DTrace

`DTrace` est un outil de traçage qui permet d'exécuter des scripts qui sont attachés à des sondes de traçage dans le noyau de Unix (Gregg and Mauro, 2011). Un utilisateur peut alors instrumenter des appels système facilement sans avoir à recompiler le code de son application. Puisque DTrace s'attache aux sondes dans le noyau, il est possible de tracer tous les processus qui s'exécutent sur le système. Originalement, `DTrace` a été conçu par Sun Microsystems pour fonctionner sur le système d'exploitation Solaris (une variante de Unix). Récemment, l'outil a été adapté pour fonctionner avec le noyau Linux par une équipe d'Oracle (Gregg, 2014a). Selon Brendan Gregg, l'auteur de DTrace Book (Gregg and Mauro, 2011), l'outil de traçage a connu son succès grâce à un effort de commercialisation accru de la part de Sun Microsystems (Gregg, 2014a).

`DTrace` est aussi compatible avec l'environnement de `Mac OS X`, étant donné que ce dernier repose sur les fondations de Unix. `DTrace` fait parti du logiciel d'aide au développement d'`Apple Inc.` nommé *Instruments* (App, 2014).

En terme de performance, le surcoût de traçage de `DTrace` augmente lors du traçage d'applications qui utilisent plusieurs fils d'exécution. L'utilisation de mécanismes de protection d'exécution concurrente lors du traçage en serait la cause (Desnoyers and Dagenais, 2012).

SystemTap

Tout comme `DTrace`, `SystemTap` permet de rédiger des scripts qui sont attachés à des sondes de traçage dans le noyau de Linux. La différence majeure avec `DTrace` est que les scripts `SystemTap` sont compilés en code natif et s'exécutent donc plus rapidement que les scripts compilés en *bytecode* de `DTrace`. Le script écrit est exécuté au début, à la fin ou dans certains cas pendant des appels système (Don Domingo, 2013). La différence majeure entre `SystemTap` et `strace` est que `SystemTap` analyse l'activité de tous les processus du système. Il est donc possible d'analyser toutes les interactions du même type pour tous les processus qui s'exécutent sur l'ordinateur, contrairement à `strace` qui analyse un ou quelques processus spécifiques. Cet atout permet l'accès à de l'information supplémentaire qui facilite alors l'analyse de données pour pouvoir trouver des problèmes causés par l'activité des autres processus.

Tout comme `DTrace` et `strace`, il n'est pas nécessaire de redémarrer le système étant donné

que la technique utilisée par `SystemTap` est non-intrusive (Cohen, 2012). Cette caractéristique permet de diagnostiquer par exemple un serveur dont la disponibilité est cruciale et qui ne doit pas être éteint.

Malheureusement, tout comme DTrace, les techniques de synchronisation utilisés par `SystemTap` font en sorte que le surcoût de traçage augmente considérablement lorsque tracé (Desnoyers and Dagenais, 2012).

LTTng

LTTng est une version réécrite de son prédécesseur *Linux Tracing Toolkit* (LTT). Ce traceur est implémenté dans l'optique de conserver la performance lors du traçage d'applications parallèles. Il enregistre ses traces dans le format *Common Trace Format* (CTF) et utilise un canal d'enregistrement par cœur de processeur. Cette approche permet de minimiser l'utilisation de primitives de synchronisation entre les fils d'exécution pour éviter une exclusion mutuelle au fichier de sortie de la trace. En utilisant un fichier par canal d'enregistrement, l'utilisation de l'exclusion mutuelle peux être évitée et la performance globale du système en souffre beaucoup moins.

Dans un autre effort de performance, l'infrastructure matérielle des processeurs est utilisée pour maximiser la performance de l'enregistrement des points de trace. Entre autres, certains processeurs *Advanced RISC Machines* (ARM) disposent d'un module *System Trace Macrocell* (STM) et son utilisation permet de réduire l'impact du traçage par un facteur de dix dans les tests faits par A. Vergé (Verge, 2014).

Pour pallier au problème que le surcoût de traçage peut imposer sur le système, LTTng enregistre les traces dans des tampons circulaires de taille fixe. Si le débit de données de traçage (producteur de trace) surpasse celui d'écriture sur le disque (consommateur de trace), le système ne bloquera pas. Par contre, quelques événements seront perdus. Cette caractéristique est très importante en ce qui concerne l'impact sur un système de production qui ne peut se permettre de cesser de s'exécuter (Desnoyers and Dagenais, 2012).

Ces plusieurs détails font en sorte que LTTng possède un surcoût de traçage moins significatif que la plupart des autres outils (Desnoyers and Dagenais, 2012) lors de scénarios de prise de traces d'exécution parallèle.

En plus de permettre la collecte de points de trace noyau, `LTTng` permet d'enregistrer des événements *User-Space Tracing* (UST) (Desnoyers, 2009). Un programmeur désirant ajouter des points de trace dans son logiciel dispose alors d'un bon moyen qui s'incorpore de façon transparente dans la trace du noyau système. Ce dernier pourra alors facilement observer

l'état du système à différents points dans l'exécution de son programme. Les deux principales façons d'ajouter un point de trace UST sont des plus simples pour le programmeur, en voici un exemple :

```
#include <lttng/tracef.h>
void someFunction() {
    [...]
    tracef("Hello UST tracef! There is an int: %d!", someInt);
    [...]
}
```

Le point de trace est alors enregistré avec une estampille temporelle du type monotonique et peut être analysé suite à son exécution.

2.1.3 Horloge

M. Desnoyers explique que l'utilisation de l'horloge monotonique est la plus stable des méthodes. Effectivement, des horloges comme celle dite "temps réel" (*CLOCK_REALTIME*) peuvent être compromises par un changement d'heure de la part d'un administrateur et alors invalider la propriété de linéarité de la trace (Desnoyers and Dagenais, 2010). L'horloge monotonique donne à l'utilisateur la garantie que sa valeur ne sera pas réduite : elle augmente constamment en fonction de compteurs matériels présents dans l'archirecture des CPU modernes. Il est important de noter que cette horloge peut être modifiée mais que l'ajustement causé par une modification ne fera que ralentir ou accélérer le compteur. Cette horloge est appropriée pour effectuer des mesures de temps d'exécution (comparaison entre deux mesures de temps monotonique) mais ne représente pas le temps depuis *epoch* (soit le 1^{er} janvier 1970). Le point de départ de l'horloge n'est pas spécifique.

2.1.4 Analyse de traces

La méthode de génération de trace et les techniques de sondage utilisées ne représentent que la portion d'acquisition de données. Une fois que ces données sont enregistrées sur disque, il faut alors trouver moyen d'interpréter ces données et de donner des outils aux programmeurs pour leur permettre de mieux cerner les problèmes potentiels de leur programme.

Une trace peut devenir très volumineuse assez vite. Des méthodes d'enregistrement et de compression sont alors recommandées pour permettre de réduire l'impact sur le système ou de réduire le taux de pertes d'enregistrement d'événements (pour les outils de trace qui utilisent cette technique). Non seulement il existe des techniques de compression spécifiques à la compression de traces d'exécution (Price and Vachharajani, 2010), mais il existe aussi des travaux

reliés à l'analyse de la performance de la compression utilisée par ces algorithmes (Stenmark, 2010).

Les traces d'exécution pouvant facilement atteindre les centaines de gigaoctets (Wininger, 2014). Il est important que le format de ces traces soit bien structuré et permette notamment de naviguer efficacement son contenu. Des outils comme LTTng enregistrent les événements dans le format CTF. Ce format consiste en une liste d'événements qui sont associés avec un estampillage temporel. La trace est linéaire, c'est-à-dire que les événements écrits dans la trace se suivent dans le temps. Cette caractéristique peut sembler évidente et logique mais la section 2.2.6 démontre en quoi elle peut devenir une limitation.

Des outils d'analyse de traces ont étés implémentés pour découvrir des patrons (*pattern*) d'exécution dans les traces (Waly and Ktari, 2011) (Ezzati-Jivan and Dagenais, 2012). D'autres sont implémentés pour exécuter des tests de validation de modèles (*model checking*) (Palnitkar et al., 1994).

2.2 Traçage logiciel d'accélérateurs graphiques

Le but principal de ce travail étant de permettre le traçage d'accélérateurs graphiques comme les GPU, nous devons maintenant aller inspecter l'état de l'art en ce qui concerne l'analyse de programmes fonctionnant sur les accélérateurs graphiques.

2.2.1 Histoire des accélérateurs graphiques

Il existe présentement deux façons (ou raisons) principales pour utiliser un accélérateur graphique. La première est la plus connue, étant donné qu'elle nous entoure constamment. Il s'agit de l'accélération 2D et 3D. L'accélération 2D/3D est la technologie qui a stimulé le besoin de matériel spécialisé comme les cartes graphiques. Dans les années 1970, les ordinateurs avaient besoin de matériel dédié pour gérer l'affichage à l'écran (Singer, 2013). Bien rapidement, les jeux vidéos ont créé une demande grandissante pour du matériel de plus en plus performant pouvant afficher plus de couleurs, plus de pixels et plus de textures. Les deux multinationales Nvidia et *Advanced Micro Devices* (AMD) sont de nos jour les principaux rivaux dans le domaine mais l'évolution rapide dans le monde de la technologie de l'information nous fait aussi connaître des nouveaux joueurs dont Intel et ARM. Les API pour l'accélération graphique sont *Open Graphics Library* (OpenGL) (une API multi-platforme) et DirectX (compatible dans Windows seulement).

C'est à partir de la moitié des années 2000 que les programmeurs se sont rendu compte qu'ils pouvaient mettre des données dans une soi-disante texture pour ensuite effectuer un calcul

graphique sur la "texture" et relire ses valeurs. C'est ainsi que la deuxième façon d'utiliser des accélérateurs graphiques est née : le calcul graphique. Ageia Technologies Inc. créa alors une plateforme parallèle : la carte PhysX. Ce périphérique était en fait un coprocesseur parallèle dédié aux calculs de la physique pour les jeux vidéos. Après avoir créé *Compute Unified Device Architecture* (CUDA), Nvidia a acheté Ageia et a adapté ses cartes graphiques pour permettre aux programmes PhysX de s'exécuter sur les cartes `GeForce` de la série 8 (NVIDIA Corp., 2015). De son côté, AMD adopta le standard *Open Source Open Computing Language* (OpenCL) comme API de calcul graphique. OpenCL étant un standard créé par `Apple Inc.` et confié sous la responsabilité de `Khronos Group`

Depuis la création des deux API de calcul graphique, OpenCL fut aussi adopté par Nvidia et plusieurs autres joueurs comme Intel. Par ailleurs, Intel produit un pilote qui porte le nom de `Beignet` (Nanhai, 2015) dont le code source est disponible et qui fonctionne avec le pilote graphique *Open Source* `i915` pour GNU/Linux.

De nos jours, les plus grosses fermes de calcul scientifique utilisent des accélérateurs graphiques comme périphérique clé dans leur infrastructure. `Titan`, le plus gros super-ordinateur américain, utilise des cartes graphiques `Nvidia Tesla K20x` (Tit, 2012). La principale raison de cette approche est la puissance de calcul par watt d'électricité consommé. Le mariage des CPU avec les GPU permet de mieux utiliser l'électricité en assignant les calculs qui sont adaptés à une architecture ultra parallèle aux cartes graphiques résultant alors en une efficacité énergétique accrue (Lavelle, 2012).

2.2.2 Intérêt de l'analyse de l'exécution GPU

Maintenant que tous les ordinateurs possèdent une carte graphique et que nous savons que certains calculs peuvent être effectués plus rapidement tout en consommant moins d'énergie, les grandes compagnies commencent à prendre un virage *"vert"*. Si une économie d'énergie peut être réalisée sur un super-ordinateur, alors c'est aussi le cas des applications sur les ordinateurs personnels et les téléphones intelligents : qui dit efficacité énergétique dit plus longue durée de la batterie pour les appareils portables.

Selon une étude commandée par Microsoft (Roth et al., 2013), la consommation électrique de son navigateur internet `Internet Explorer 10` sur le système d'exploitation `Windows 8` serait moins grande que ses compétiteurs. Le directeur de la commercialisation d'`Internet Explorer`, Roger Capriotti, affirme que cette efficacité énergétique est étroitement liée à l'utilisation accrue des accélérateurs graphiques lorsque possible (Capriotti, 2013).

2.2.3 Langages de calcul graphique

La section précédente (2.2.1) discute des API de calcul graphique CUDA et OpenCL. Ces deux interfaces de programmation procurent aussi un langage de programmation spécifique pour les noyaux de calcul qui sont exécutés sur les GPU. Dans les deux cas, il s'agit d'un langage très semblable au `C`. Les noyaux de calcul sont les programmes qui sont exécutés sur les accélérateurs graphiques.

CUDA

CUDA étant une API propriétaire, il ne peut pas fonctionner sur les accélérateurs graphiques qui ne sont pas manufacturés par Nvidia. Par contre, il est important de noter qu'OpenCL est supporté par les cartes Nvidia. Pour cette raison et aussi dans l'optique de ne pas restreindre la portée de ce mémoire aux cartes Nvidia, CUDA ne sera pas l'objectif principal de cet ouvrage.

Le fonctionnement de cette technologie est très semblable à celui d'OpenCL, fonctionnement qui est décrit dans la section suivante (section 2.2.3).

OpenCL

L'API d'OpenCL offre deux types de fonctions.

1. Fonction synchrone
2. Fonction asychrone

Les fonctions synchrones permettent d'obtenir des informations quant à la configuration du système et de préparer les noyaux d'exécution, arguments et autres structures de données nécessaires à la bonne exécution du programme. Ci-dessous se trouve une liste de ces principales structures de données ainsi que leur nom en anglais (qui est utilisé dans l'API).

- Système (*Platform*)
- Périphérique (*Device*)
- Contexte (*Context*)
- File de commande (*Command Queue*)
- Tampon (*Buffer*)
- Programme (*Program*)
- Noyau (*Kernel*)
- Événement (*Event*)
- ...

Le processus d'utilisation d'OpenCL consiste en la configuration d'une file d'exécution dans un contexte configurable sur un périphérique de calcul d'un système (Khronos Group Inc.,

2011). Des programmes appelés "noyau" sont généralement compilés en fonction de l'architecture de l'accélérateur graphique (périphérique) sur lequel le noyau s'exécutera.

Par la suite, des commandes sont ajoutées à la file d'exécution. Ces commandes impliquent principalement des requêtes d'entrée/sortie de données et des requêtes d'exécution de noyau. Ces requêtes sont asynchrones, c'est-à-dire que l'appel de fonction de l'API retourne après avoir ajouté la requête dans la file de commandes et non pas lorsque la commande finit de s'exécuter. La figure 2.1 représente l'architecture de programmation de l'API OpenCL.

File de commandes

Les cartes graphiques exécutent des commandes qui sont ajoutées en file d'exécution. Dans le pilote de la carte graphique se trouve une file de commandes qui réside dans l'hôte. Les commandes qui s'y trouvent sont en attente d'être transférées vers la seconde file de commandes qui se trouve sur la carte graphique.

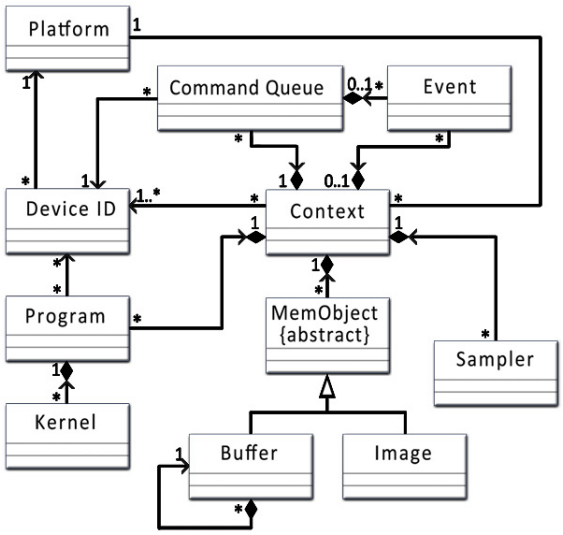
L'ordonnancement des commandes de la file de commandes est crucial. C'est en grande partie la responsabilité de l'utilisateur de bien configurer les commandes afin de tirer avantage des différentes capacités du périphérique. Par exemple, si le calcul ne dépend pas des résultats précédents, on peut réduire le goulot d'étranglement (souvent la communication de données entre l'hôte et le périphérique) en effectuant le transfert de données en même temps que l'exécution d'un noyau sur la carte. Il est important de prendre en considération que l'ordonnanceur des commandes sur le périphérique reçoit aussi les commandes des autres programmes qui utilisent l'accélérateur graphique. L'effort de synchronisation que le programmeur met pour maximiser l'utilisation de toutes les ressources de l'accélérateur graphique peut être infructueux si un autre programme ajoute des commandes dans la file de commandes en même temps.

La section 2.2.5 discute des outils de traçage disponibles et de leurs limitations quant au traçage du système global.

Usage des accélérateurs graphiques

Dans le monde du calcul haute performance, il n'est pas rare que certains algorithmes doivent effectuer le même calcul pour chaque élément d'un tableau de données. Avant les *General Purpose Graphical Processing Unit* (GPGPU), la seule solution pouvant accélérer ces calculs sur un système était de paralléliser l'exécution sur tous les cœurs du CPU de ce dernier. Avec un accélérateur graphique, ce calcul peut être exécuté sur des centaines, voir des milliers de cœurs simultanément. Les accélérateurs graphiques offrent plus de deux niveaux de magnitude

Figure 2.1 Diagramme de classe d'OpenCL 1.2



Source : Khronos Group Inc. (2011)

de parallélisation de plus que les CPU modernes.

2.2.4 Architectures d'accélérateurs graphiques

La principale différence entre le processeur central et le GPU est que le processeur graphique peut effectuer des opérations *Single Instruction Multiple Data* (SIMD) à une échelle hors norme comparé aux instructions SIMD du CPU. Les plus récentes cartes graphiques, au moment de l'écriture de ce mémoire, ont plus de 2 000 cœurs de calcul. Ces cœurs de calcul fonctionnent généralement à une fréquence quelque peu moins rapide que sur un CPU et la complexité des instructions sur GPU fait en sorte que ces dernières peuvent nécessiter plusieurs instructions pour accomplir l'équivalent d'une seule instruction sur un CPU (pour les instructions CPU complexes). Néanmoins, la quantité de cœurs de calcul outrepasse généralement cette différence.

Il existe deux formats d'accélérateurs graphiques :

1. Dédiés : Dont le processeur ne fait pas partie de la même puce électronique que le CPU mais est plutôt une carte d'expansion qui communique via le bus Peripheral Component Interconnect *(PCI) Express* (PCIe).
2. Intégré : Dont le processeur graphique est intégré à la même puce de silicium que le processeur central.

Les accélérateurs graphiques dédiés possèdent de la mémoire *Graphics Double Data Rate* (GDDR). Cette mémoire est utilisée comme mémoire globale pour l'accès aux données et

instructions d'un programme. Elle joue le même rôle que la mémoire *Double Data Rate* (DDR) du processeur central mais possède une architecture différente pour permettre un débit d'écriture adapté à l'infrastructure hautement parallèle des cartes graphiques.

La plupart des accélérateurs graphiques intégrés, dont les `Intel HD Graphics` et les AMD *Accelerated Processing Unit* (APU), font usage de la mémoire partagée (la mémoire DDR du processeur central). Cette caractéristique fait en sorte que la communication entre l'hôte et la carte graphique subit une latence moins grande.

Il s'agit ici du portrait global des accélérateurs graphiques. En fait, il existe des variations qui sont dépendantes des implémentations, en fonction des différentes architectures des différents manufacturiers. Le tableau 2.1 illustre un exemple des différentes spécifications des architectures des différents vendeurs.

Tableau 2.1 Spécifications techniques de GPU modernes

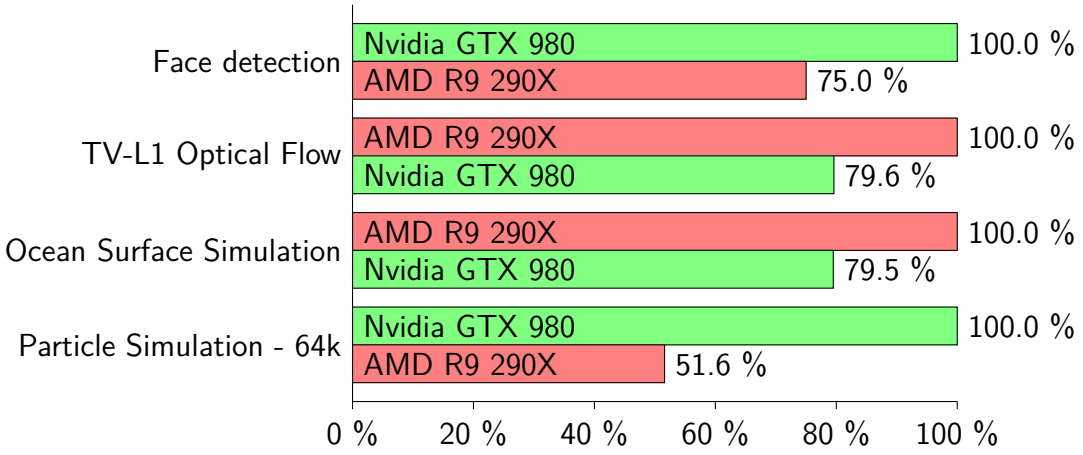
Vendeur	Type	Modèle	Nbr. Cœurs	Fréq.(MHz)	Mémoire (GB)
AMD	Dédié	R9 290X	2 816	1 000	8.0
Nvidia	Dédié	GTX 980	2 048	1 126	4.0
Intel	Intégré	Iris Pro Graphics 5200	40	1 300	Partagé +0.125 Caché

Il est important de noter que les différences architecturales significatives entre les produits des deux grands manufacturiers (AMD et Nvidia) font en sorte qu'on ne peut comparer les spécifications entre les deux compétiteurs et assumer qu'un GPU de la compagnie `X` est plus performant que celui de la compagnie `Y` en se référant au nombre de cœurs et aux fréquences. En fait, bien que la carte AMD possède plus de cœurs de calculs, dans certains scénarios c'est l'architecture de Nvidia qui offre la meilleure performance. On peut observer différentes mesures de performance relative selon la tâche dans la figure 2.2 qui illustre des données de performance mesurées et distribuées par (Kishonti Ltd., 2015). Ces résultats démontrent bien qu'une architecture n'est pas nécessairement meilleure qu'une autre dans toutes les situations.

AMD

AMD utilise présentement l'architecture *Graphics Cores Next* (GCN) pour ses cartes graphiques. Une des particularités de cette architecture est l'utilisation de plusieurs unités SIMD. Le noyau de l'architecture, le processeur de commandes GCN, agit comme interface primaire entre les différents types de commandes : les commandes graphiques et les commandes d'exécution de nuanceurs ("*shaders*" en anglais) (AMD, 2012).

Figure 2.2 GPU : Performance relative en fonction du type de travail



Source des données : (Kishonti Ltd., 2015)

Lorsque l'API d'OpenCL fait des appels de fonctions pour soumettre des tâches, il communique en fait avec une couche de pilote graphique qui prend en charge la gestion des commandes. AMD offre une garantie : les commandes mises en file seront exécutées dans l'ordre dans lequel elles ont été reçues. Comme mentionné dans la section 2.2.3, les appels à l'API OpenCL qui constituent des commandes asynchrones doivent être ajoutés à une file d'attente. Dans le cas d'AMD, le contenu de cette file d'attente réside initialement sur l'hôte avant d'être transféré sur le périphérique. La file d'attente est alors prise en charge par le matériel de l'accélérateur graphique (AMD, 2013, sect. 2.4.1).

Nvidia

Nvidia vient tout juste de changer d'architecture pour `Maxwell`. La principale différence architecturale comparativement à son prédécesseur `Kepler` est que le rapport unité de contrôle en fonction du nombre de cœurs de calcul a été augmenté. Nvidia essaie ici de permettre une meilleure performance pour les plateformes qui font une utilisation concurrente de la carte graphique (Harris, 2014). Plusieurs contextes d'exécution pourront alors être exécutés simultanément sur un seul périphérique à la fois. Cette fonctionnalité fait en sorte que l'utilisation des ressources sera maximisée dans les cas où un programme n'utilise pas tous les cœurs de calcul du périphérique. De là l'importance de pouvoir tracer l'exécution du système en entier pour mieux comprendre le comportement global.

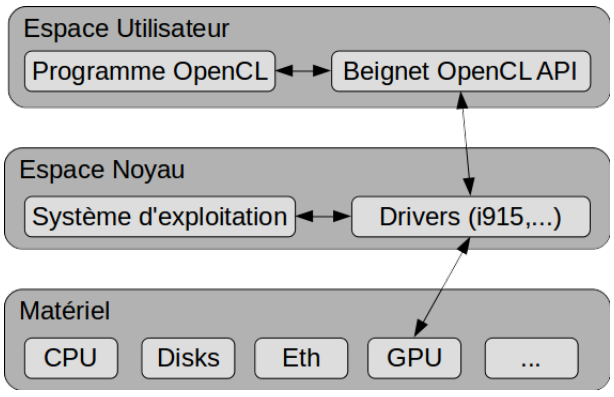
Intel

Suite à l'avènement des nouvelles technologies et des avantages reliés aux accélérateurs graphiques, Intel a commencé à produire des GPU intégrés de meilleure qualité : `Intel HD Graphics`. Son prédécesseur, `Intel` ***`Graphics Media Accelerator`*** `(GMA)`, n'était pas intégré à la même puce de silicium que le processeur central mais plutôt dans le "*northbridge*" de la carte mère. Suite à une ré-implémentation de la structure du processeur central et des méthodes de communication, le *chipset* en charge des communications (le *northbridge*) fut intégré au processeur central, y ramenant alors le composant d'accélération graphique qui s'y trouvait.

`Intel HD Graphics` représente maintenant une très bonne solution pour l'utilisation de tous les jours et peut même supporter quelques jeux vidéos exigeants avec une bonne performance. Une équipe de programmation d'Intel travaille sur le pilote supportant OpenCL sur ces accélérateurs graphiques qui se nomme `Beignet`. La beauté de ce projet est qu'il est à code source libre et fonctionne par-dessus les pilotes graphiques d'Intel qui sont aussi à code source libre. Il est donc possible d'accéder à toutes les ressources nécessaires pour analyser le fonctionnement complet de l'accélérateur graphique.

Comme le montre la figure 2.3, `Beignet` consiste en une librairie qui s'exécute au niveau utilisateur. Elle fait le lien entre l'application de l'utilisateur qui désire utiliser OpenCL et les pilotes graphiques `i915` d'`Intel`. Ces derniers, qui se situent dans l'espace d'exécution noyau, communiquent les requêtes au matériel. Dans le cas échéant, il s'agit du GPU `Intel`.

Figure 2.3 Position de Beignet dans les espaces d'exécution



2.2.5 Outils disponibles

C'est plutôt dans le domaine de la prise de traces pour le calcul haute performance sur accélérateurs graphiques (aussi connu comme GPGPU) que l'information académique est rare à trouver. La principale raison est que les principaux outils d'analyse sont propriétaires aux grandes compagnies, les deux plus grands joueurs étant AMD et Nvidia. Cependant, une des exceptions est le logiciel de traçage *Tuning and Analysis Utilities* (TAU) (TAU, 2013) : il offre aussi des mesures de performance pour les applications qui utilisent les accélérateurs graphiques et utilisent entre autres la librairie propriétaire *CUDA Profiling Tools Interface* (CUPTI) de Nvidia pour prendre ses mesures (Juckeland, 2012). Ce logiciel peut aussi agir comme enveloppe à la librairie OpenCL : méthode que nous avons aussi utilisée et qui sera discutée plus tard dans la section 2.2.6.

AMD

AMD supporte l'API OpenCL sur ses cartes graphiques dédiées, sur ses cartes graphiques intégrées ainsi que sur ses CPU. L'outil de traçage qui est mis à la disposition des programmeurs dans Linux et Windows est `CodeXL`. Cette application est plutôt complète en ce qui concerne le traçage et le débogage des applications qui utilisent le GPU. En plus d'agir comme une couche entre le programme analysé et la librairie OpenCL, afin d'enregistrer les appels de fonction de l'API, l'outil utilise l'API `GPUPerfAPI` pour extraire des données de performance supplémentaires quant à l'exécution des noyaux et les transferts de données. Ces métriques de performance permettent entre autres d'analyser le pourcentage d'occupation des cœurs de calcul lors de l'exécution d'un noyau, permettant alors au programmeur d'ajuster les dimensions de calcul utilisées en paramètre pour maximiser l'utilisation du périphérique qu'il cible.

En plus de permettre de tracer l'utilisation du périphérique, `CodeXL` permet aussi de déboguer un programme OpenCL à l'aide de points de trace dans le code du noyau d'exécution.

Par contre, ce dernier ne permet pas de visualiser l'exécution des autres programmes qui utilisent la carte graphique simultanément à l'exécution du programme spécifié. Son analyse reste donc restreinte, dans les cas où les ressources de calcul graphique sont partagées. Aussi, les appels système sont manquants dans la trace du programme. Le programmeur ne peut donc pas avoir accès à toutes les données nécessaires pour pouvoir comprendre le système en entier.

La figure 2.4 illustre un exemple de trace d'exécution . Elle consiste en une trace texte qui commence par une petite section de *metadata*, suivie de la liste des appels de fonctions de

l'API OpenCL et qui termine par les données d'estampillage de temps des appels de fonction de l'API, des transferts de données et des exécutions de noyaux. Comme vous pouvez le voir,

Figure 2.4 Exemple d'enregistrement de trace CodeXL

```
[...] (Metadata: contenu peu important)
=====AMD CodeXL ocl API Trace Output=====
5916
651
CL_SUCCESS = clEnqueueWriteBuffer ( 0x0588BFD8;0x05F18C70;CL_TRUE;0;1024;0
    x023AC138;0;NULL;NULL )
[...] (autres appels de fonction OpenCL)
=====AMD CodeXL ocl Timestamp Output=====
5916
651
54    clEnqueueWriteBuffer                          21530198448541
    21530229236101        4596     CL_COMMAND_WRITE_BUFFER
    21530198562784        21530198568864        21530229226418
    21530229229381        2          0x0588BFD8                2          0
    x09FB6D68                Tahiti                          1024
[...] (autres estampillages de temps)
```

cette trace est composée de texte et utilise un format standard de longueur par lignes qui est bourré par des espaces vides si nécessaire. Dans la section d'estampillage de temps, selon nos mesures, les nombres affichés semblent concorder avec la mesure du temps de l'horloge monotonique du système. La provenance de cette mesure lorsqu'elle représente les temps d'exécution sur la carte graphique n'est pas spécifiée dans la documentation du manufacturier. Sa similitude avec l'horloge monotonique suggère deux possibilités :

1. Au démarrage, l'horloge du périphérique est synchronisée entre l'hôte et le périphérique : cette lecture est alors mesurée sur le périphérique.
2. L'horloge n'est pas synchronisée : cette lecture provient donc de l'horloge monotonique de l'hôte.

Nvidia

Nvidia fournit aussi une suite complète de développement pour leur API propriétaire CUDA. Cette suite d'outils est aussi compatible avec OpenCL et se nomme `Nsight`. Cet outil est disponible en *plugin* dans Eclipse et Microsoft Visual Studio. Tout comme `CodeXL`, `Nsight` permet non seulement de tracer l'exécution sur les accélérateurs graphiques mais aussi de déboguer un programme qui en fait l'utilisation. Des points de débogage peuvent être insérés dans le programme.

Nvidia permet aussi d'accéder à des données de compteurs de performance tout comme `GPU PerfAPI` d'AMD. L'API disponible pour obtenir ces données est CUPTI. En plus de permettre l'accès à ces données, on peut s'inscrire à des rappels de fonctions (*callback*) pour obtenir des données de traçage et les estampilles de temps d'exécution sur la carte.

L'enregistrement de la trace faite par `Nsight` est encodé pour sauver de l'espace dans un fichier de type *Extensible Markup Language* (XML). Cette technique permet de réduire le surcoût relié au traçage, présentant donc un avantage comparé à la méthode d'enregistrement utilisée par `CodeXL`.

Intel

Plusieurs outils d'analyse de la performance sont disponibles pour analyser les processeurs Intel. Malheureusement, ce sont tous des logiciels commerciaux fermés. L'outil principal se nomme `VTune` : il est l'outil principal pour analyser et optimiser les programmes qui utilisent les ressources Intel. Il permet entre autres de profiler les CPU, cartes `Xeon Phi` (un coprocesseur d'architecture x86 hautement parallèle) et accélérateurs graphiques. L'outil permet d'analyser l'exécution des programmes OpenCL seulement à partir du système d'exploitation Windows (Intel Corp., 2015).

Outils non propriétaires

Plusieurs outils non propriétaires permettent d'obtenir des traces d'exécution de programmes qui utilisent l'accélération graphique fournie par OpenCL. Les plus connus dans le domaine sont `TAU` et `VampirTrace`. Ces deux outils sont à code source libre.

`TAU` est un outil d'analyse de la performance qui est développé par une équipe de l'Université d'Oregon (TAU, 2013). Il permet lui aussi d'enregistrer les appels aux librairies de calculs OpenCL et CUDA. De plus, il utilise les APIs de mesure de performance *GPU PerfAPI* et CUPTI pour enregistrer des métriques de performance propres aux infrastructures d'AMD et de `Nvidia` respectivement.

`VampirTrace` est compatible avec la librairie de calcul parallèle *Message Passing Interface* (MPI). Pour ce faire, il offre au programmeur un compilateur spécial qui instrumente les appels à l'API dans le fichier binaire de l'exécutable. L'utilisateur qui désire tracer l'exécution d'un programme doit avoir le code source de ce dernier et le recompiler en utilisant leur compilateur. Le support pour intégrer l'analyse de l'exécution de programmes utilisant le GPU a aussi été réalisé.

2.2.6 Méthodologie utilisée

Tracer les accélérateurs matériels est un sujet qui a déjà été exploré. G. Jukeland en dédie un chapitre complet dans son livre de traçage haute performance (Juckeland, 2012). Ce type de traçage implique un niveau de complexité bien différent du traçage système classique et plusieurs aspects dont la prise de métriques temporelle et les méthodes d'interception d'appel aux librairies sont discutés.

Méthodes de traçage

Les solutions amenées par G. Jukeland sont aussi présentes dans l'infrastructure des outils de profilage disponibles, notamment en ce qui concerne l'instrumentation dynamique des programmes utilisant la librairie OpenCL. Grâce à son code source libre, il est possible de confirmer que `TAU` utilise les méthodes décrites par G. Jukeland. Nous pouvons soulever l'hypothèse que les outils propriétaires en font de même.

Le principe consiste à instrumenter les appels à la librairie pour intercepter les appels à cette dernière et y insérer des points de trace. Il est aussi question de la méthode d'enregistrement des événements asynchrones (exécution de noyaux de calcul et transfert de données entre l'hôte et le périphérique) qui nous permet de s'abonner à un rappel de fonction et d'obtenir les informations dans un tampon de trace associé avec l'événement, après la fin de l'événement en question.

Synchronisation des horloges

Comme discuté dans la section 2.2.4, un autre problème qui découle du traçage de coprocesseurs est le fait que l'horloge du coprocesseur n'est pas nécessairement synchronisée avec celle du processeur central. Des méthodes de synchronisation proposées par B. Poirier et R. Roy (Poirier et al., 2010) sont présentement utilisées dans des outils d'analyse de traces comme `Trace Compass` pour pouvoir synchroniser des traces qui ont lieu sur des ordinateurs différents communiquant via le réseau.

Obtention de données de traçage de façon asynchrone

Comme la section 2.1.3 le mentionne, l'utilisation d'une horloge monotonique incrémentale est nécessaire pour assurer une propriété de linéarité des événements dans le temps. En d'autres mots, un événement qui s'est produit dans le passé ne peut pas être enregistré dans la trace. Un défi avec les traces de périphériques est que l'enregistrement des événements ne peut pas

être fait à partir du périphérique lui-même. Il est vrai que le périphérique peut estampiller un événement mais, dans tous les cas, il ne s'agira pas de la même horloge que celle utilisée pour le système, c'est-à-dire l'horloge monotonique.

En utilisant des techniques de synchronisation de traces, on peut être en mesure de pouvoir synchroniser les horloges de l'hôte et du périphérique, mais le problème principal ne sera toujours pas réglé. Effectivement, OpenCL permet d'obtenir des métriques de temps d'exécution pour les exécutions de fonctions asynchrones comme l'exécution d'un noyau OpenCL ou le transfert de données entre l'hôte et le périphérique. Dans l'API existant, quatre différents estampillages de temps sont disponibles via la fonction *clGetEventProfilingInfo* :

1. *CL_PROFILING_COMMAND_QUEUED* :
   La requête a été ajoutée dans la file d'attente de l'hôte
2. *CL_PROFILING_COMMAND_SUBMIT* :
   La requête a été transférée de la file d'attente de l'hôte à celle du périphérique
3. *CL_PROFILING_COMMAND_START* :
   La requête a commencé à s'exécuter sur le périphérique
4. *CL_PROFILING_COMMAND_END* :
   La requête a fini de s'exécuter sur le périphérique

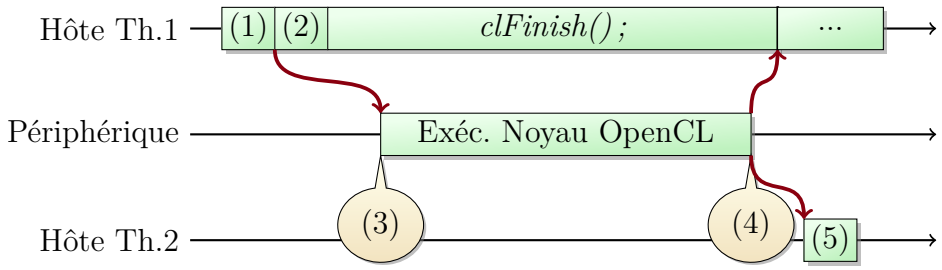
Étant donné que les estampillages de temps sont ajoutés en utilisant un compteur qui se situe sur le périphérique et que l'exécution se passe sur le périphérique, l'événement de début de l'exécution ne peut être obtenu qu'à la fin de l'exécution. La figure 2.5 illustre une technique qui est disponible dans l'API d'OpenCL et qui consiste à enregistrer une fonction de rappel qui sera exécutée après la fin de l'exécution de la requête. Cette fonctionnalité est disponible grâce à la fonction *clSetEventCallback* qui prend entre autres en paramètre l'événement auquel est attachée la requête.

En utilisant ces métriques de temps, nous avons assez d'information pour être en mesure de savoir en tout temps, pour le processus analysé, quel est le contenu de chaque file de commandes (de l'hôte et du périphérique), le (ou les) noyau(x) qui s'exécute(nt), ainsi que les transferts de données entre l'hôte et le périphérique.

Le problème qui complique la compatibilité avec les outils de prise de traces comme LTTng est que les outils fournis par la librairie UST de LTTng ajoutent l'estampillage de temps monotonique automatiquement au moment où l'appel à la fonction de traçage est fait. L'insertion d'événements plus tôt dans la trace n'est alors pas possible.

`CodeXL` semble utiliser les mêmes techniques de traçage que celles décrites dans la section 2.2.6. Pour contourner le problème de la prise de traces asynchrones, il semble que les données

Figure 2.5 Profilage du temps avec l'API d'OpenCL



(1) : *clEnqueueNDRangeKernel(..., &event) ;* (lancement du noyau de calcul)
(2) : *clSetEventCallback(event, ...) ;* (enregistrement d'une fonction de rappel pour l'événement)
(3) : Temps de début du calcul sur le périphérique
(4) : Temps de fin du calcul sur le périphérique
(5) : Fonction de rappel qui fait la réception et traitement des données de temps

de la trace soient conservées en mémoire jusqu'à ce que le traçage soit fini, étant donné le caractère de la trace physique qu'on peut observer dans la figure 2.4. Pour chaque événement, l'appel à la fonction est enregistré dans la première section de la trace, puis les métriques temporelles sont enregistrées dans la section qui suit. Pour les événements qui ont interagi avec la carte graphique (exécution de noyaux ou transferts de données), des données de traçage supplémentaires sont disponibles. Ces informations sont entre autres les différents temps que la fonction *clGetEventProfilingInfo* peut nous fournir. Une grande trace nous montrera que la lecture de façon linéaire de cette trace est impossible, étant donné que les événements d'exécution de noyaux et de transferts de données génèrent des données de temps asynchrones.

Un bon exemple serait d'imaginer la trace que la figure 2.5 produirait (les points de trace qui ci-dessous se suivent dans la trace sur disque et $[a, b, c, d, e, f, g, h]$ sont des points dans le temps qui se suivent) :

1. Début de la fonction *clEnqueueNDRangeKernel(...) ;* au moment a
2. Fin de la fonction *clEnqueueNDRangeKernel(...) ;* au moment b
3. Début de la fonction *clSetEventCallback(event, ...) ;* au moment c
4. Fin de la fonction *clSetEventCallback(event, ...) ;* au moment d
5. Début de la fonction *clFinish() ;* au moment e
6. Fin de la fonction *clFinish() ;* au moment h
7. Début de l'exécution du noyau (point (3)) au moment f
8. Fin de l'exécution du noyau (point (4)) au moment g

Cet exemple illustre bien que la trace aurait dû être enregistrée sur disque dans l'ordre suivant : [1, 2, 3, 4, 5, **7, 8, 6**] afin d'être considérée comme linéaire.

Il est important de comprendre que les outils d'analyse de traces CPU sont implémentés pour être en mesure de traiter des traces souvent de taille énorme. Des méthodes de navigation sont implémentées dans ces outils pour permettre de parcourir rapidement les traces et la linéarité est assumée. Le comportement asynchrone de la prise de traces GPU fait en sorte que l'utilisation de ces outils d'analyse de traces est impossible dans l'état où ils sont présentement implémentés, sans un post-traitement pour réordonnancer les événements.

2.3 Conclusion de la revue de littérature

Cette revue de littérature nous a permis d'explorer les technologies existantes en ce qui concerne le traçage système conventionnel. Plus précisément, nous avons souligné la différence entre les niveaux d'exécution ainsi que les outils disponibles qui permettent le traçage des différents niveaux d'exécution. Les caractéristiques des plateformes de traçage furent discutées et les avantages de celles-ci présentés en offrant une comparaison entre eux. Le sujet important de la source de temps nous permis de prendre conscience de l'importance d'une source de temps monotonique pour associer des estampillages temporels aux événements tracés. Aussi, la section d'analyse de traces nous a permis de prendre connaissance du format d'enregistrement CTF qu'utilise LTTng.

La seconde section de la revue de littérature nous a présenté les caractéristiques des accélérateurs graphiques modernes, l'architecture ultra parallèle et les outils qui permettent l'obtention de traces pour les programmes bénéficiant d'accélération graphique.

Malheureusement, nous pouvons voir qu'aucun outil de traçage n'offre présentement la possibilité d'acquérir une trace unifiée de l'exécution système et graphique en une seule trace.

CHAPITRE 3 MÉTHODOLOGIE

Dans le chapitre précédent, nous avons présenté les outils, technologies et défis existants pour tracer le CPU et le GPU. L'introduction quant à elle, nous a énuméré des objectifs spécifiques. Parmi ces derniers, on observe entre autres la présence d'objectifs de performance. La mesure de performance requiert une méthodologie stricte et bien documentée afin d'en permettre la reproductibilité. C'est dans cette optique que nous vous présenterons les détails de prises des métriques de performance qui furent utilisées dans cette recherche.

Les détails de la configuration matérielle et logicielle vous seront partagés dans la section 3.1. Ensuite, des explications détaillées vous expliqueront comment utiliser la librairie de traçage OpenCL CLUST qui est présentée dans le Chapitre 4 de ce mémoire. Par la suite, les détails des logiciels utilisés et de la méthode utilisée pour obtenir les données de surcoût qui seront présentés dans l'article sont expliqués. Finalement, des liens vers les dépôts du code source de l'utilitaire de génération ainsi que de la solution proposée sont donnés dans la dernière section de ce chapitre.

3.1 Environnement de travail

Dans cette section, nous discuterons plus en détail de la configuration de l'ordinateur utilisé pour relever les données de mesure de performance qui sont présentées dans l'article du chapitre 4 et dans le chapitre d'analyse complémentaire (chapitre 5). De plus, nous discutons de la source de temps utilisée pour relever ces données.

3.1.1 Station de tests

Lors de l'exécution des tests, le poste informatique utilisé fut toujours le même afin d'obtenir des données de performance les plus précises possible. Puisque les processeurs `Intel` possèdent désormais une carte graphique assez évoluée et que les pilotes de ceux-ci sont à sources ouvertes sur GNU/Linux, c'est cette plateforme qui fut principalement analysée. Le fait que le processeur graphique soit situé sur la même puce informatique aide aussi à minimiser l'impact des latences des transferts de données entre l'hôte (le CPU) et le périphérique (le GPU), étant donné que la communication entre l'hôte et le périphérique ne doivent pas passer par le bus PCIe. En considérant que le coût de traçage est constant par rapport à la taille du problème et la quantité de cœurs de calcul disponible sur le GPU, les mesures de surcoût par fonctions devraient être semblables pour les GPUs qui possèdent plus de cœurs

de calcul.

Les spécifications du poste informatique de test sont les suivantes :

1. Processeur central Intel i7-4770
2. Processeur graphique intégré Intel HD4600
3. 32 GB DDR3 RAM 1600 MHz

En plus de la configuration matérielle utilisée, il est important de spécifier la configuration logicielle qui fut utilisée pour effectuer les mesures :

1. Ubuntu 14.04 Desktop avec noyau version 3.18.4
2. LTTng version 2.6.0-rc1
3. Librarie OpenCL `Beignet` v1.0.2 avec modifications faites (commit git #3df61a4... 6ee18d9)
4. Mesures effectuées alors que tous les autres processus étaient fermés.

3.1.2 L'horloge utilisée pour les tests

Pour acquérir les données de performance, parmi tous les choix de sources de temps, c'est l'horloge monotonique du système qui fut utilisée. Cette horloge garantit la propriété de monotonicité, c'est-à-dire qu'une lecture faite après une autre donnera forcément une valeur de retour plus grande que la précédente. Cette horloge permet aussi une fine granularité au niveau des nanosecondes.

3.2 Traçage avec CLUST

3.2.1 Définition de la variable d'environnement LD_PRELOAD

La librairie de traçage CLUST existe sous le format d'un fichier de librairie appelé *"libCLUST.so"*. Le principe de fonctionnement est de replacer les symboles de la librairie OpenCL au chargement qui précède le lancement de l'application à tracer. Pour ce faire, l'utilisateur qui désire tracer une application OpenCL doit placer en chargement dynamique la librairie CLUST en assignant la variable d'environnement *LD_PRELOAD* avant de lancer le programme à tracer.

Pour ce faire, l'utilisateur peut assigner la variable en se limitant à la portée du processus :

```
1 LD_PRELOAD=/path/to/clust/dir/libCLUST.so ./monProgrammeOpenCL
```

Aussi, il est possible d'exporter la variable d'environnement au niveau de la session :

```
export LD_PRELOAD=/path/to/clust/dir/libCLUST.so
./monProgrammeOpenCL
```

Finalement, afin de tracer toutes les applications du système qui utilisent OpenCL, l'utilisateur peut aussi modifier le fichier des variables d'environnement globale (*/etc/environment* sur Ubuntu 14.04) pour y ajouter la définition de la variable d'environnement *LD_PRELOAD*.

3.2.2 Activation de la prise de traces LTTng-UST

Bien que les symboles des fonctions OpenCL soient redirigés par la librairie CLUST quand cette dernière est spécifiée en chargement dynamique (section 3.2.1), le traçage ne s'effectue que quand LTTng enregistre les événements UST.

Pour y arriver, l'utilisateur doit créer une trace LTTng, activer tous les points de trace LTTng-UST pour la librairie CLUST et démarrer l'enregistrement de la trace :

```
lttng create clustTrace
lttng enable-event --userspace "clust_provider:*"  # Tous les points de trace
    spécifique à CLUST
lttng start
```

Le traçage commence pour tous les processus du système qui utilisent OpenCL, et qui ont leurs symboles interceptés par la librairie CLUST, quand la commande `lttng start` est lancée. Pour arrêter le traçage, il suffit de faire un appel à `lttng stop`.

3.2.3 Définition du surcoût

Le surcoût est le terme utilisé pour définir le temps supplémentaire qu'induit l'utilisation de CLUST par rapport au temps normal sans l'utilisation de CLUST. Dans le cas de CLUST, nous pouvons définir deux types de surcoût : le surcoût de préchargement et le surcoût de traçage.

Surcoût de préchargement

Le remplacement dynamique des symboles d'OpenCL par ceux de la librairie CLUST implique que CLUST doit imiter le fonctionnement de la librairie OpenCL en interceptant les appels dédiés à cette librairie pour instrumenter ces appels de fonction.

Même quand LTTng ne trace pas les points de trace CLUST, pour chaque appel de fonction OpenCL, au moins deux points de trace sont faits pour tracer le début et la fin de l'appel de

fonction. Bien que ces points de trace ne soient pas activés, ils ajoutent une charge supplémentaire qui doit être mesurée. Les sections 4.5.2, 4.5.3 et 4.5.4 présentent une mesure de ce surcoût.

Surcoût de traçage

Le surcoût le plus significatif est celui lorsque LTTng enregistre les événements LTTng-UST que la librairie CLUST génère. Encore une fois, ce surcoût est présenté dans les sections 4.5.2, 4.5.3 et 4.5.4 de l'article.

Il représente la différence de temps entre l'exécution normale d'un programme qui utilise OpenCL et le temps d'exécution du même programme alors qu'il est tracé par la librairie CLUST.

3.2.4 Logiciel utilisé pour mesurer le surcoût

Afin de bien mesurer le surcoût de la librairie CLUST, nous avons fait plusieurs différents scénarios de mesure :

1. Évaluation du surcoût pour les appels de fonction simple
2. Évaluation du surcoût pour les appels de fonction qui exécutent une tâche asynchrone sur le GPU
3. Évaluation du surcoût pour l'exécution d'un programme OpenCL simple

Pour ces trois scénarios, nous avons mesuré le temps d'exécution normal, le temps d'exécution alors que les symboles d'OpenCL sont surchargés par CLUST mais que le traçage n'est pas activé et le temps d'exécution alors que LTTng enregistre les points de trace que CLUST génère.

Ces différentes mesures nous permettent d'évaluer facilement l'impact du traçage sur une variété de différents programmes en connaissant simplement le nombre d'appels synchrones et asynchrones effectués par itération du programme OpenCL.

Mesure du surcoût des fonctions synchrones

Pour les fonctions OpenCL qui effectuent des appels dit *synchrones*, c'est-à-dire qui ne créent pas de commande asynchrones pour le GPU, l'algorithme suivant fut utilisé :

```
1 clock_gettime(CLOCK_MONOTONIC, &start_clock);
2 for(iter = 0; iter < benchMax; iter++) {
3     clGetPlatformIDs(0, NULL, &num_platforms);
```

```
4 }
5 clock_gettime(CLOCK_MONOTONIC, &end_clock);
```

Un objectif collatéral à la détermination du surcoût était de prouver que LTTng pouvait supporter une grande quantité de points de trace sans toutefois ralentir le système. Pour ce faire, des échantillons de taille fixe de 1 000 itérations furent effectués et la variable `benchMax` fut modifiée afin d'augmenter la charge et d'en mesurer le comportement.

Mesure du surcoût des fonctions asynchrones

Pour les fonctions OpenCL qui engendrent des commandes au GPU, une fonction bloquante de lecture fut utilisée. Étant donné que cette fonction bloque jusqu'à la fin de la lecture, elle est idéale pour prendre des mesures. Autrement, un appel à une fonction de synchronisation de temps comme *clFinish* ou *clWaitForEvent* aurait été nécessaire. On aurait alors mesuré l'exécution d'une fonction synchrone en plus de celle asynchrone. Ainsi, le code utilisé est le suivant :

```
1 clock_gettime(CLOCK_MONOTONIC, &start_clock);
2 for(iter = 0; iter < benchMax; iter++) {
3     clEnqueueReadBuffer(queue, mem_c, CL_TRUE, 0, buffer_size*sizeof(cl_uint),
      local_c, 0, NULL, NULL);
4 }
5 clock_gettime(CLOCK_MONOTONIC, &end_clock);
```

Étant donné que nous avons déjà déterminé la mise à l'échelle de LTTng dans les mesures de la section précédente (3.2.4), ici nous ferons varier la taille du tampon transféré afin de calculer l'impact de l'utilisation de CLUST en fonction de la charge de travail effectuée.

Mesure du surcoût du programme SimpleOpenCL

Afin de bien représenter le surcoût lors de l'exécution d'un vrai programme, nous présentons aussi des mesures de performance qui prennent en compte le pipeline normal d'OpenCL en présentant l'impact en fonction de la charge de travail accomplie.

Le programme utilisé est assez simple : à chaque itération, il configure les paramètres du noyau OpenCL, effectue un calcul arithmétique en fonction des paramètres fournis et l'hôte effectue une lecture du tampon de sortie. Étant donné qu'ici c'est la performance après l'initialisation qui nous intéresse, nous mesurons uniquement le temps nécessaire pour 100 itérations de calcul. La quantité d'échantillons récoltés est de 1 000 : cela nous permet d'obtenir un écart type assez petit.

3.3 Code source CLUST

Pour automatiser l'instrumentation de la librairie OpenCL, un programme Java fut écrit, il prend en paramètre la liste de fonctions OpenCL et génère automatiquement les fichiers nécessaires pour la librairie CLUST. Il est alors facile d'apporter des modifications à toutes les fonctions OpenCL et de supporter différentes versions de la librairie. Le code est disponible sur `GitHub` à l'adresse suivante : `https://github.com/dcouturier/LibraryOverrider.git`.

La librairie CLUST est aussi à sources ouvertes, le dépôt de code est accessible sur `GitHub` à l'adresse suivante : `https://github.com/dcouturier/CLUST.git`.

CHAPITRE 4 ARTICLE 1 : LTTng CLUST: A system-wide unified CPU and GPU tracing tool for OpenCL applications

Authors
David COUTURIER <david.couturier@polymtl.ca>
École Polytechnique de Montréal

Michel R. DAGENAIS <michel.dagenais@polymtl.ca>
École Polytechnique de Montréal

Keywords: OpenCL, Tracing, Profiling, GPGPU, LTTng

Submitted to: Hindawi: Advances in Software Engineering, April 14^{th} 2015

4.1 Abstract

As computation schemes evolve and many new tools become available to programmers to enhance the performance of their applications, many programmers started to look towards highly parallel platforms such as GPU. Offloading computations that can take advantage of the architecture of the GPU is a technique that has proven fruitful in recent years. This technology enhances the speed and responsiveness of applications. Also, as a side effect, it reduces the power requirements for those applications and therefore extends portable devices battery life and helps computing clusters to run more power efficiently.

Many performance analysis tools such as `LTTng`, `strace` and `SystemTap` already allow CPU tracing and help programmers to use CPU resources more efficiently. On the GPU side, different tools such as `Nvidia`'s `NSight`, `AMD`'s `CodeXL` and third party `TAU` and `VampirTrace` allow tracing API calls and OpenCL kernel execution. These tools are useful but are completely separate and none of them allow a unified CPU-GPU tracing experience.

We propose an extension to the existing scalable and highly efficient `LTTng` tracing platform to allow unified tracing of GPU along with CPU's full tracing capabilities.

4.2 Introduction

Tracing programs has been a common technique from the beginning. Tracers such as DTrace (Gregg and Mauro, 2011), strace (Fagan, 1998), SystemTap (Don Domingo, 2013) and LTTng (Desnoyers, 2009) have been around for many years. They allow recording not only kernel trace events but some also allow recording user space events. This is a very good alternative to debuggers for finding software bugs and especially performance related issues that concern the execution on the CPU side. The rapid changes in computation never stop to bring new challenges. As part of those rapid changes, we have seen GPU acceleration become a more common practice in computing. Indeed, the different highly parallel architectures of the graphic accelerators, compared to the conventional sequential oriented CPU, represent a very attractive tool for most of the graphical work that can be achieved on a computer. This became the standard in recent years. Most operating systems have even added a graphic accelerator as a hardware requirement. Indeed, its processing power is being harnessed in many of the operating system's embedded tools and help make computations more fluid, therefore enhancing user experience. The evolution of classical graphical acceleration in games and user interfaces brought architecture changes to the GPU. The graphical accelerators evolved from the graphical pipeline to a more programmable architecture. Programmers quickly figured how to offload raw computations to the graphical accelerator and GPU computing, also referred as *General Purpose Graphical Processing Unit*s, was born. APIs such as Nvidia [1] CUDA [2] and Apple's open standard OpenCL(Khr) (developed by the Khronos Group) were created and adapted to standardize the usage of GPUs as GPGPUs. As part of this change, many tools were implemented to help programmers better understand how the new architecture handles the workloads on the GPGPU. The two major manufacturers, AMD [3] and Nvidia provided their analysis, tracing and debugging tools: CodeXL [4] and NSight [5] respectively.

Unfortunately, none of the GPGPU tracing tools are currently capable of seamlessly providing both a comprehensive GPU trace alongside with a fully capable CPU oriented trace, that tracers such as DTrace, strace, SystemTap or LTTng already provide. Also, current tools do not allow for a system-wide tracing of all processes using the GPU: current techniques require the user to manually specify what processes to trace.

In this article, we present a solution to acquire a system-wide trace of OpenCL GPGPU

1. http://www.nvidia.com/
2. https://developer.nvidia.com/cuda-zone
3. http://www.amd.com/en-us/products/graphics
4. http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/
5. http://www.nvidia.com/object/nsight.html

kernel executions for a better understanding of the increasingly parallel architecture that GPUs provide. This solution is also presented as an extension of the already existing CPU tracer LTTng to allow for unified CPU-GPU trace gathering and analysis capabilities.

In addition, we take a look at kernel trace point analysis for the open source Intel i915 drivers that support OpenCL, and for which an open source library for OpenCL, called *Beignet*[6], is provided.

This article is divided in four main parts. First, in Section 4.3, we explore the CPU and GPU tracing options and methods that are currently available. Then, in Section 4.4, the architecture, implementation choices and algorithms are explained, followed by a performance analysis of the solution proposed in Section 4.5. Finally, we conclude and outline possible future work in Section 4.6.

4.3 Related Work

Tracing is the act of recording events during the execution of a program at run-time. This is commonly used with not only GNU/Linux ((strace project, 2010), (Don Domingo, 2013), (Desnoyers, 2009)) but also all major operating systems such as Unix (Gregg and Mauro, 2011) and Windows [7]. The main purpose is to provide a tool for the developers to record some logs about the execution of programs and detect complex problems such as race conditions.

In this section, we first present the different available tracing platforms that are available. Secondly, we discuss what kind of information trace analysis provides, followed by current GPU tracing architectures, tracing tools, and OpenCL tracing capabilities. Then, the topic of time keeping is discussed: many different timing metrics are available and they need to be explored. Moreover, the device timing metrics that can be accessed from the OpenCL library on the GPU may not be using the same source as the host: synchronization methods to help match those two separate sources will be discussed (Section 4.3.4). Finally, we talk about tracing in the device drivers as a complementary measure of OpenCL tracing.

4.3.1 CPU Tracing

Tracing on GNU/Linux is somewhat easier than doing it on Windows or other proprietary operating systems. One good reason for that is because we have access to its source code. Since our work focuses on the GNU/Linux operating system, we will direct our attention to this environment and explore available tools on this platform. The major tools identified are

6. https://wiki.freedesktop.org/www/Software/Beignet/
7. https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803.aspx

strace (Fagan, 1998), DTrace (Gregg and Mauro, 2011), SystemTap (Don Domingo, 2013) and LTTng (Desnoyers, 2009).

strace

The main benefit of strace (Fagan, 1998) is its ease of use and the fact that it has been around since the early 90's. The strace functionality allows the user to record all or some specific system calls of a given program or a list of processes. The output is either sent to *stderr* or redirected to a file. The content of the output is the name of the recorded system call with its parameters and the return value of the call. For instance, recording only the file open and close operations of a program can be done by executing the following command: `strace -e open,close ./myProgram`.

Despite the fact that strace is easy to use, its most obvious limitation lies in the fact that it uses a single output file for trace recording and therefore degrades performance when tracing multithreaded applications since it has to use *mutexes* to write from the multiple threads to the single file. This issue is discussed in (Desnoyers and Dagenais, 2012).

Strace also has a hidden performance weakness: when tracing an application, the API used to get the system call information (*ptrace*) adds an overhead of two system calls per system call that the traced program performs (Gregg, 2014b). This can have a dramatic impact on the system due to the known overhead of system calls. Therefore, the strace approach does not seem to be suitable in the long run.

DTrace

DTrace (Gregg and Mauro, 2011) originated from the Solaris operating system. Solaris is a Unix variant developed by Sun Microsystems which was later bought by Oracle. After the acquisition of Sun Microsystems, Oracle ported the tracing tool to the GNU/Linux *Operating System* (OS) (Gregg, 2014a). DTrace relies on hooking scripts to probes that are placed in the operating system kernel. Those scripts can then process and record system calls or other information. A script is compiled into *bytecode* in order to allow fast execution through a small and efficient kernel based virtual machine. DTrace also provides a library called *User Statically Defined Tracing* (USDT) that programmers can use to instrument the code of their applications and perform *user space* tracing (Gregg and Mauro, 2011).

One major issue with DTrace is its usage of synchronization mechanisms that produce a large tracing overhead when tracing heavy workloads such as multithreaded applications (Desnoyers and Dagenais, 2012). Since programs nowadays are mostly multithreaded, this is a serious

limitation.

SystemTap

Like DTrace, SystemTap (Don Domingo, 2013) provides a framework for hooking scripts to probes in the kernel. It also offers user space tracepoint capabilities and may qualify as a system-wide tracing tool. The main difference with DTrace is that it originated on GNU/Linux and the scripts are compiled to native code instead of bytecode.

While SystemTap allows for user space tracing, a feature needed for OpenCL tracing, the compiled tracepoint script is executed in kernel space. This forces the system to go back and forth from user space to kernel space in order to execute the script. This adds significantly to the cost of executing tracepoints and leads to higher overhead.

Again, like DTrace, SystemTap suffers considerably from the synchronization mechanisms used: creating a large overhead when tracing heavy applications (Desnoyers and Dagenais, 2012). For the same reason as DTrace, heavily multithreaded applications will suffer from a very large overhead with SystemTap.

LTTng

Kernel space tracing for LTTng (Desnoyers, 2009) is also based on hooking to *kprobes* and trace points. The approach to get the information is different from DTrace and SystemTap: modules are written in native *C* and compiled to executable code. LTTng was designed with the idea of tracing multithreaded applications with low overhead: each CPU thread has separate output buffers in order to minimize the need for synchronization methods when writing the trace events. As they are generated, before being written to disk, the trace events are stored in a circular buffer that has a specified capacity. In the worst case, if the event data throughput exceeds the disk throughput, events are intentionally discarded, rather than blocking the traced application, in order to minimize the tracing overhead.

User space event tracing is also provided with the *LTTng-UST* library and, unlike SystemTap, the user space tracepoint is recorded in a different channel entirely in user space. This allows for minimal tracing overhead (Goulet, 2012). As it will be discussed in Section 4.3.3, tracing OpenCL can be achieved at the user space level. The advantages of LTTng over the other available options justifies its choice as a tracing platform for this project.

4.3.2 Trace Analysis

Getting the trace information is only one part of the tracing procedure. The encoding used to record traces and the potentially huge size of traces impose strict requirements on the analysis tools. For instance, LTTng provides viewing tools such as *babeltrace*[8] and *Trace Compass*[9] that allow decoding binary traces to human readable format (*babeltrace*) and interactive visualization and navigation (*Trace Compass*).

Visualization tools provide a very good understanding of program execution and allow for quick overall understanding of potential execution bottlenecks and synchronization issues. More advanced analysis can also be performed on traces, for example pattern searching algorithms for uncovering specific execution patterns within traces (Waly and Ktari (2011), Ezzati-Jivan and Dagenais (2012)), model checking (Palnitkar et al., 1994) and statistics computation (Ezzati-Jivan and Dagenais (2013), Ezzati-Jivan and Dagenais (2015)).

4.3.3 GPU Tracing

Research teams around the world have spent a great amount of time developing applications to enable programmers to have a better understanding of the GPU kernel execution (Dietrich et al., 2012; Mistry et al., 2011). It is thus relevant to examine GPU architectures. GPUs started as specialized hardware designed to deal with the display monitor. It first evolved into a graphical pipeline of operations, and then towards the more programable highly parallel architecture that we have today (Owens et al., 2008).

Many programming APIs were implemented to support GPGPU but Nvidia's CUDA and the open standard API OpenCL (supported on several different hardware architectures) were the result of need for high level API (Owens et al., 2008). Both APIs provide functions to harness the computing power of graphics cards.

The GPU highly parallel architecture allows not only faster but also more energy efficient operations. This characteristic leads to longer battery life for portable devices and better performance per watt[10].

GPU architecture

The reason behind the efficiency of GPUs compared to CPUs in highly parallel situations is that the GPU offers hundreds and even thousands of smaller computation cores built

8. https://www.efficios.com/babeltrace
9. http://projects.eclipse.org/projects/tools.tracecompass
10. http://top500.org/list/2014/11/

under the SIMD architecture. Not all computations can take advantage of such architecture. This is why only parts of programs can be run on GPUs. The SIMD architecture allows multiple different data elements to be processed simultaneously under the same computing instruction (Owens et al., 2008).

Different form factors are available: integrated graphics are very popular in the portable computing world but dedicated graphics cards still hold a significant computing power advantage. AMD and Nvidia both offer integrated graphics and dedicated graphics devices, while Intel only offers integrated graphics. All architectures differ a lot from one companie to another and also between models. The performance may vary depending on the application: some architectures are better suited for some kind of computations and vice versa.

Computing on GPUs is a technique that derived from a workaround on the existing capabilities of available hardware. The library that makes OpenCL work on GNU/Linux is a library that positions itself between the program and the drivers but still operates in user space. Figure 4.1 shows the position of Intel's open source OpenCL library for GNU/Linux called *Beignet*. Computing on the GPU with OpenCL is done asynchronously: commands are sent

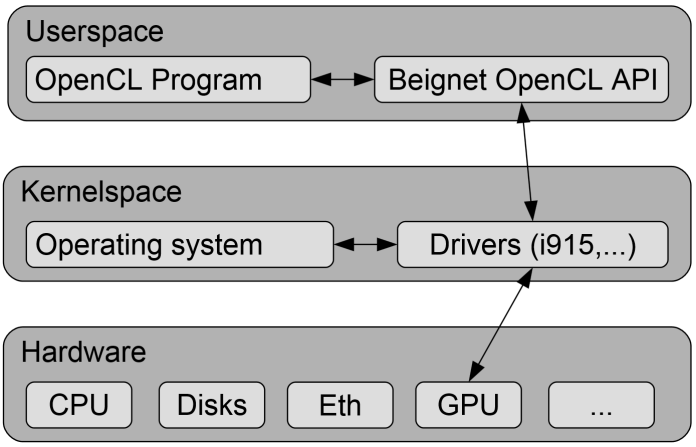


Figure 4.1 Beignet OpenCL library position in the system

to the GPU driver command queue and, when the device is ready to receive commands, they are transferred from the host driver command queue to the device command queue. The driver is notified when the commands are done executing.

Available tools

As discussed in Owens et al. (2008), tracing tools for GPUs are primarily proprietary tools provided by the manufacturers:

1. AMD: CodeXL
2. Nvidia: Nsight
3. Intel: VTune

Third party tools such as *TAU*[11] and *VampirTrace*[12] provide much of the same information as the proprietary ones but also add support for other computing APIs such as MPI.

The information that these tools can provide is the arguments and timing of the API calls to the OpenCL library, as well as some instrumentation that Juckeland (2012) presents in order to get timing for the execution of the events on the graphics card, using profiling tools that OpenCL can provide. CodeXL, Nsight, VTune, TAU and VampirTrace provide a visualization of the API calls, with related OpenCL context and GPU activity (memory transfers and kernel execution on the device) underneath the API calls.

None of these tools provide a system-wide analysis of all activities related to a specific device (GPU). Also, they do not feature system tracing information that could help provide a more accurate picture of the execution context when debugging. Our solution addresses this issue by proposing a system-wide tracing technique that integrates with LTTng for simultaneous GPU and system tracing.

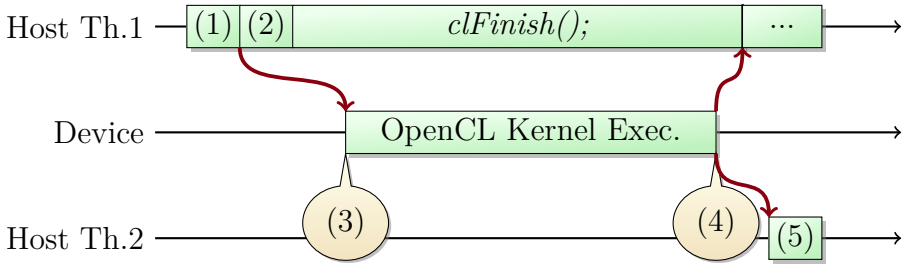
OpenCL profiling capabilities

As described by Juckeland (2012), tracing the execution of a program is fairly straightforward. OpenCL provides a function to gather information about the execution of a kernel or a data transfer: we will refer to those events as *asynchronous commands*. This information has to be accessed asynchronously after the asynchronous command completion. The easiest way to collect this information is to subscribe to a callback function that is defined for the event associated with the asynchronous command, using the *clSetEventCallback* function. As Figure 4.2 shows, the timing information that concerns the start and end of the kernel execution can be accessed after the end of the kernel execution (at point (5) in Figure 4.2). For the information to be accessed, the programmer shall use the *clGetEventProfilingInfo* function. Many different timing metrics can be accessed from this function. Here is a list of available information and their description from the open source specification (Khronos Group Inc., 2011):

1. Command Queued:
   The command has been added to the host queue

11. https://www.cs.uoregon.edu/research/tau/home.php
12. http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace

Figure 4.2 OpenCL API Profiling



(1): *clEnqueueNDRangeKernel(..., &event);* (computing kernel launch)
(2): *clSetEventCallback(event, ...);* (subscription to a callback for the end of the kernel execution)
(3): Start time of the computation on the device
(4): End time of the computation on the device
(5): Callback function that records the information ((3),(4) and other)

2. Command Submitted:
   The command has been transferred from the host queue to the device queue
3. Command Execution Start:
   The command started executing on the device
4. Command Execution End:
   The command is done executing on the device

All of those metrics allow access to a 64 bits integer device time counter at the moment specified (Khronos Group Inc., 2011).

Knowing these four time specifications, we are now able to define precisely the content of the three waiting zones where the commands can reside:

1. On the host driver waiting queue
2. On the device waiting queue
3. Executing on the device.

Being able to understand the content of the queues allows the programmer to understand his program resource usage. Then, he can modify how his program interacts with the device and enqueues commands in order to maximize computation resources usage. For instance, instead of waiting during data transfers, it is often feasible to overlap computations on one set of data with the transfers for retrieving the previous set and sending the next set.

4.3.4 Time Keeping

The importance of the source time clock selection is exposed in Desnoyers (2009). Not all clocks have the monotonic property and, in order to ensure that all events are recorded sequentially as they happen, the use of a monotonic clock is the key. The real time clock for instance can get set to a earlier time by an administrator or by *Network Time Protocol* (NTP). Using the *C* function `clock_gettime(CLOCK_MONOTONIC,[...]);` to access the time reference helps preserve the monotonicity of the trace.

It is important to understand that this clock does not have an official start point. It only guarantees that a reading will not have a timestamp that precedes past readings. Its usage is restricted to comparing events timing within the same computer session, since the timer is reset to an unspecified value at reboot time.

The same principle is also valid for GPU time stamps: they rely on on-chip hardware to get the time and, as the CPU monotonic clock, this time stamp only allows for the computation of the difference between two time stamps (the time delta) taken from the same device. Again, this time stamp origin does not refer to any specific time and is therefore different from the CPU time stamp. Synchronization methods are required to match the CPU event time stamp to the GPU time stamp. This problem was already discussed and a synchronization solution was suggested by Poirier et al. (2010) for synchronizing network traces. The same method may be applied to GPU tracing.

4.3.5 i915 Tracing

Another way to trace GPU activity it to record already existing trace points that are present in the Intel *i915* GPU drivers. This approach allows monitoring the commands on the host queue and sent to the device, without the need for an interception library such as the solution presented in this paper: CLUST. The upside of this technique is that it monitors all the GPU activity, including graphics (OpenGL).

One disadvantage of this technique is that it does not allow modifying the applications execution and, as Table 4.1 shows, the only data available is the current content of the host side and device side queues, as well as commands that are being executed on the device.

The reason why *i915* monitoring does not allow getting command-specific timing metrics is because in CLUST, in order to access a command start and end time, a profiling flag is enabled when a command queue is created. This leads to extra event data being pushed to the GPU and filled with timing data on the GPU side (events that are invisible to the GPU drivers).

Table 4.1 Recording capabilities differences between CLUST and GPU driver monitoring

	CLUST	i915 monitoring
Host-side command queue	✓	✓
Device-side command queue	✓	*
Command-specific metrics	✓	×

*: We know that a command is either in the device's command queue or currently executing on the device

Therefore, tracing *i915* drivers would be more beneficial as a complement of tracing with CLUST. This would allow getting a trace that monitors all GPU activity on the host side, and also geting the more detailed information regarding OpenCL that CLUST provides.

4.4 Implementation

In this section, we present the implementation details of our solution that addresses the identified challenges (Section 4.3), followed by a system-wide tracing solution and ways to visualize the OpenCL trace.

4.4.1 CLUST Implementation

As discussed in Section 4.3.3, the OpenCL API can be divided into two separate function types: synchronous and asynchronous:

1. Synchronous API calls:
   Calls that do not enqueue tasks on the GPU
2. Asynchronous API calls:
   Calls that enqueue tasks on the GPU

Those two types of functions require a different approach for recording relevant tracing metrics.

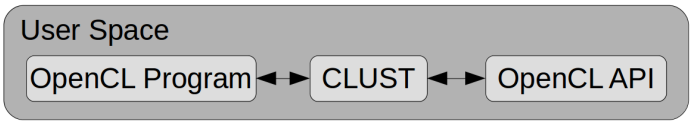
Synchronous OpenCL tracing

As seen in Section 4.3.3, the OpenCL library operates in user space. The purpose of CLUST is to trace an application without the need for recompiling it from its source code. Therefore, CLUST is a library that wraps all function calls of the OpenCL library and automatically instruments all the calls to the API. Figure 4.3 shows how the API calls are intercepted by the CLUST library. The library file, named *libCLUST.so*, has to be preloaded using `LD_PRELOAD` before launching the program. In the constructor of the library, the OpenCL

symbols are loaded from the real OpenCL library and assigned to function pointers that have the same name with the `reallib_` prefix. Then, all OpenCL function are redefined, and the start and end of the API calls are recorded using LTTng-UST:

```
cl_int clWaitForEvents(cl_uint num_events, const cl_event *event_list) {
    tracepoint(clust_provider, cl_clWaitForEvent_start, num_events);
    cl_int ret = reallib_clWaitForEvents(num_events, event_list);
    tracepoint(clust_provider, cl_clWaitForEvent_end);
    return ret;
}
```

Figure 4.3 CLUST position within the user space



Asynchronous OpenCL tracing

This type of recording works well for synchronous function calls. However, for asynchronous function calls such as kernel command enqueue and data transfer command enqueue, a different approach must be considered. All of the asynchronous commands have an event provided as parameter. This parameter may be `NULL`, when the caller does not intend to use an event. When `NULL`, because CLUST internally needs it, an event has to be allocated and later freed in order to avoid memory leaks. Here is an example with the asynchronous function *clEnqueueWriteBuffer*:

```
cl_int clEnqueueWriteBuffer(cl_command_queue [...], cl_event * event)  {
    // Check if device tracing enabled?
    const bool trace = __tracepoint_clust_provider___clust_device_event.state;
    bool toDelete = event == NULL;
    if(caa_unlikely(trace)) {
        if(toDelete) { // Dynamic event allocation
            event = malloc(sizeof(cl_event));
        }
    }

    tracepoint(clust_provider, cl_clEnqueueWriteBuffer_start, [...relevant
    arguments to record...]);
    cl_int ret = reallib_clEnqueueWriteBuffer(command_queue, buffer,
    blocking_write, offset, cb, ptr, num_events_in_wait_list, event_wait_list,
    event);
```

```
13      tracepoint(clust_provider, cl_clEnqueueWriteBuffer_end);
14
15      if(caa_unlikely(trace)) {
16          int r = reallib_clSetEventCallback(*event, CL_COMPLETE, &
    eventCompleted, (toDelete)?&ev_delete:&ev_keep);
17          if(r != CL_SUCCESS) { [...error management...] }
18      }
19
20      return ret;
21  }
```

Since this function only enqueues a command in the command processor of the driver, we have to wait until the end of the command execution to get access to more valuable timing metrics. The callback function *eventCompleted* will be called at the end of the command execution. This is where the interesting timing metrics can be accessed. We access the four timing metrics discussed in Section 4.3.3 using the *reallib_clGetEventProfilingInfo* function and, since we do not know which queue and command type the callback is referring to, we also access the command type id and the command queue id that are associated with the event, using the *reallib_clGetEventInfo* function. All collected information is then recorded in a LTTng UST trace point and the *event* is freed, if the callback parameter specifies that it has to be.

Unfortunately, the timing data collected in this way cannot be written to the trace in sequential time order, since this information was accessed after the execution of the command. This leads to problems at trace analysis time since existing tools assume that the events are written to the trace in a sequential time order.

In addition to recording API function call start and end times, the tracepoints can include relevant metrics to allow the analysis of other important information. This relevant data can be important for later analysis and allow for derived analysis, such as calculating the throughput of data transfers by saving the size in bytes of the data being transferred. By associating this measurement with the timing data recorded by the asynchronous method described previously, we can obtain the throughput of each data transfer. Other metrics such as the queue id can allow for a better understanding of the queues content.

4.4.2 System-Wide OpenCL Profiling

One of the main advantages of CLUST over existing tracing tools is that it can be used to profile not only one process at a time but also all concurrent OpenCL activity simultaneously.

This can be achieved by forcing *LD_PRELOAD* of the *libCLUST.so* library directly from

the */etc/environment* variable definition file. Every program will inherit the overridden OpenCL symbols and then all OpenCL applications would automatically be traced when LTTng's CLUST tracing is enabled. The results section (Section 4.5) show that the overhead of CLUST when the library is preloaded but not tracing is very small. Therefore, this could even be used in production, for 24/7 operation, without significant impact on system performance.

4.4.3 Non-Monotonic Tracepoint Management

Previously, we identified two different types of CLUST events: the synchronous and asynchronous events. As seen in Section 4.4.1, tracing synchronous events is simple and the timestamp linked to the events is added by LTTng-UST using the monotonic clock at the time of the tracepoint execution.

The challenge lies in collecting asynchronous timestamp data; since the device event timing metrics are accessed after the execution of the command, the timestamp linked with the LTTng-UST event shall not be used to display the device event timing. The timestamps reside in the payload of the tracepoint, and the payload timing data that has to be used.

The trace analysis tools, by default, use the tracepoint timestamp for display and analysis. Special treatment would thus be required to indicate that in fact an event contains information about four asynchronous events, each with its own timestamp. Furthermore, these embedded asynchronous event timestamps may not appear sequentially in the trace, and use a different time source. Analysis tools such as *Trace Compass*[13] are designed to deal with very large traces and process events for display and analysis in a single pass, assuming a monotonic progression of timestamps. Different approaches can be used to solve this problem; their implementation is discussed in the next section.

Writing in different trace channels

LTTng records traces in different channels, depending on the logical CPU id, in order to record traces with low overhead. Using additional channels for asynchronous OpenCL events would be possible. The delay between the synchronous events and the asynchronous events could be tolerated in this way. However, this would only work if the asynchronous events are always executed in submission order. One channel would be created for each time value type (Kernel queued, Kernel submitted, Kernel execution start, Kernel execution end). If the submission order is always followed, and asynchronous callback events are received in the

13. http://projects.eclipse.org/projects/tools.tracecompass

same order, the events could be written in each channel when the callback is received and would remain in sorted time order in each channel. However, since the command scheduling on the device is not necessarily first-in first-out, this technique cannot be used.

Sorting

If there is an upper bound on the extent of reordering needed among commands, it is possible to buffer the events in memory and not output events before every preceeding event has been seen. When information about an asynchronous command is obtained through a callback, four events are described. By correlating this information with the *clEnqueueNDRangeKernel* calls, it would be possible to determine when all data from earlier events has arrived and it is safe to write some of these events to the trace.

Figure 4.2 depicts the execution of a call to enqueue an OpenCL kernel and its execution on the device. Ignoring the calls to *clSetEventCallback* that are not traced (action performed by CLUST), the order in which the trace is written to the file is the following:

1. *clEnqueueNDRangeKernel* start
2. *clEnqueueNDRangeKernel* end
3. *clFinish* start
4. *clFinish* end
5. Kernel execution timing metrics: queued time, submit time, start time and end time

The reprocessing of the trace would extract the four events and require reordering the tracepoints as follows:

1. *clEnqueueNDRangeKernel* start
2. Kernel queued to the host's driver
3. Kernel submitted to the GPU
4. *clEnqueueNDRangeKernel* end
5. *clFinish* start
6. Kernel execution start
7. Kernel execution end
8. *clFinish* end time

When a command is queued with *clEnqueueNDRangeKernel*, the corresponding event can be used to mark the appearance of that kernel, k_i. The callback event for k_i is received later with the execution timing metrics. The latest kernel queued with *clEnqueueNDRangeKernel*

at that time, k_j, where $j > i$, is noted. The four embedded events in the k_i callback (Kernel queued, Kernel submitted, Kernel execution start, Kernel execution end), are then buffered until they can be written safely to the trace because it can be verified that all information about earlier events has been received. This happens when all callbacks for events up to k_j have been received. Indeed, any kernel starting after k_j cannot have any of its four embedded events with timestamps earlier than those in k_i.

The number of such events that can be *in flight* is bounded by the total size of the queues (in the library, in the driver and on the card). This buffering and proper sorting could be achieved at trace generation time, when receiving callbacks in CLUST. Alternatively, it could be a separate post-processing step, or be performed at trace reading time in *Trace Compass* as an input filter. Doing this at trace generation time, rather than as a separate step, may be more efficient overall, because it avoids writing and later reading and reprocessing. However, minimizing the traced system perturbation is the first priority, which makes this solution less attractive.

If the upper bound on the number of events to buffer for sorting is too large, one can decide to resort to general sorting, using one of the available algorithms for external sorting, such as fusion sort. The downside is then having to reprocess the whole trace file before being able to visualize it. This also does not comply with live trace analysis requirements. At least, the asynchronous events should be recorded in their own channel, to separate them from the other sorted events and minimize the size of the trace file needing a reordering step.

4.4.4 GPU-CPU Trace Synchronization

As explored in Section 4.3.4, the timing sources of the CPU trace and the GPU trace are not the same. For this reason, the time relation between both time sources needs to be evaluated. This will allow aligning properly the CPU trace with the GPU timing metrics collected.

To compute the time relation, we must find the OpenCL API enqueue functions within the CLUST LTTng-UST trace and determine the most closely related device event tracepoint:

1. The queued time (GPU) of the asynchronous command is after the event time of the OpenCL enqueue API call (CPU).
2. The end time of the asynchronous command (GPU) is before the event time (CPU) for the callback providing the device events timing metrics.

These two inequalities can then be used to obtain the linear relation between the two timing sources by applying the Convex Hull synchronization algorithm, as proposed in Poirier et al.

(2010), or the more efficient improvement, suitable for live streaming, proposed in Jabbarifar (2013). In order to perform trace synchronization, we will need to ensure that all relevant information is gathered in the trace. LTTng-UST already records the *cpu_id* and time stamp (CPU) linked to the OpenCL API call start and end tracepoints as well as the callback event tracepoint. The additional information required in the tracepoint event payloads is the callback event pointer in order to properly match the *clEnqueueNDRangeKernel* start tracepoint with the corresponding callback event tracepoint.

4.4.5 OpenCL Trace Visualization

The main visualization tool for LTTng is `Trace Compass` (previously known as the *Tracing and Monitoring Framework* plugin within the Eclipse Linux Tools project). This tool already provides visualization for LTTng kernel space tracing and user space tracing. As shown in Figure 4.4, the left tree displays a hierarchy of processes. The principle is to link the OpenCL activity as a child to the associated process. This implies that there should be one child per thread performing API calls to OpenCL, and two additional children per context linked to the parent: one for displaying data transfers between the host and the device, and the other one for displaying the OpenCL kernel execution. Figure 4.4 shows how the trace unification enhances the information shown to users.

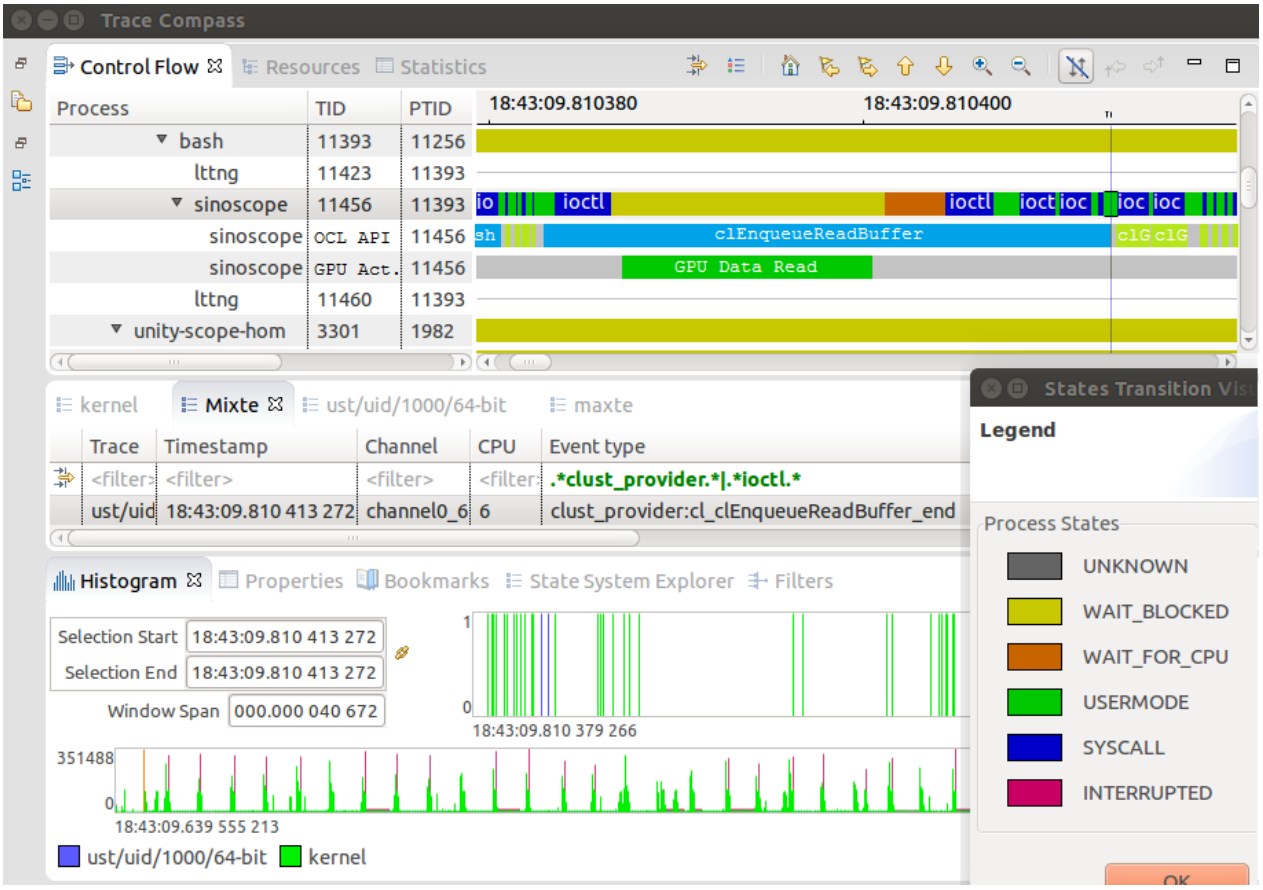
In addition to the *Control Flow* view, the *Resources* view can be used to display GPU usage: two lines per GPU: one for the data transfers and the second one for OpenCL kernel execution.

It is important to consider that the architecture of modern GPUs allow for concurrent multi-kernel execution on the same device, via different streams. If the GPU has multiple streams, it then requires more than one row for displaying OpenCL kernel execution.

4.5 Results

In this section, we discuss the results and primarily the overhead of using the OpenCL tracing library CLUST. First, we expose the details of our test platform and the different tests configurations that will be exercised. Then, we measure the library's overhead for both types of tracepoints. Secondly, we measure the impact of tracing on a real OpenCL program. Finally, examples of how the combined CPU and GPU traces can help troubbleshoot issues and enhance application performance will be presented and discussed.

Figure 4.4 Trace Compass GPU trace integration example



4.5.1 Test Setup

To demonstrate the low impact of CLUST and LTTng-UST tracepoints, we benchmarked different scenarios of execution with the following different factors considered:

1. Sample size

   We measured 100 samples of different sizes ranging from 1 call to 100M OpenCL synchronous function calls. The need for different sample sizes is to test whether the recording of many tracepoints overwhelm the system or if the design of LTTng really allows for maximum performance.

2. Benchmark configuration

   In order to establish a baseline and isolate the impact of the different components, we measured three different execution configurations:

   (a) Base: Tracing disabled, without CLUST

   This will serve as the reference for our overhead tests.

   (b) Preload: Tracing disabled, with CLUST preloaded

When replacing the symbols of the OpenCL library, an overhead is added: the two disabled LTTng-UST tracepoints and the indirection of the pointer to the actual OpenCL function.

(c) Trace: Tracing enabled with CLUST preloaded
This is the same as the preload but with tracing enabled: the overhead of storing and recording to disk the events will be added to the preload overhead.

3. Traced element type
As discussed earlier, there are two types of functions in the OpenCL API: the synchronous API functions that do not require any GPU interaction and the asynchronous API functions that enqueue commands to interact with the GPU. We suspect that the impact of CLUST for asynchronous functions is higher since a callback is hooked to every asynchronous OpenCL function calls.

We show in Tables 4.2, 4.3 and 4.4 the average time in nanosecond for each benchmark configurations (*Base*, *Preload* and *Trace*) as well as the standard deviation. The overhead displayed is the difference between the measurement and the *Base*.

The benchmarks were performed on our test platform running an Intel i7-4770, 32 GB of memory and using the integrated GPU (HD4600) with the *Beignet* OpenCL development drivers. Tracing was done with LTTng (v2.6.0) and run on Ubuntu Desktop 14.04 LTS with an upgraded kernel (v3.18.4).

4.5.2 Synchronous Function Tracing Overhead

Table 4.2 Synchronous OpenCL API function overhead benchmark

Loop Size	Base Ave. (ns/call)	Base Std Dev. (ns/call)	Preload Ave. (ns/call)	Preload Std Dev. (ns/call)	Trace Ave. (ns/call)	Trace Std Dev. (ns/call)	Preload Overhead (ns/call)	Trace Overhead (ns/call)
1	16	3	18	3	383	8	2	367
10	5.2	0.5	7.8	0.6	366.5	2.2	2.6	361.3
10^2	4.64	0.04	6.66	0.05	365.68	6.38	2.02	361.04
10^3	4.291	0.006	6.058	0.028	365.168	2.88	1.767	360.877
10^4	4.277	0.012	6.283	0.036	359.780	13.425	2.006	355.503
10^5	4.526	0.005	6.484	0.101	359.379	1.055	1.958	354.853
10^6	4.531	0.029	6.467	0.097	363.313	5.138	1.936	358.782
10^7	4.537	0.018	6.499	0.150	361.145	2.791	1.962	356.608
10^8	4.535	0.022	6.460	0.026	361.108	1.966	1.925	356.573

Sample size = 100

For the synchronous API calls, the data is displayed in Table 4.2. The conclusion we can observe from these results are the following:

1. The average base time (*BaseAverage*) for a call to clGetPlatformID is about 4.5 ns
2. The average "preloaded" time (*PreloadAverage*) for a call to clGetPlatformID is about 6.5 ns
3. Since each "preloaded" call makes two UST tracepoint calls (without recording anything), we can calculate that the average time for a disabled UST tracepoint (T_1) is only 1 nanosecond:

$$T_1 = \frac{PreloadAverage - BaseAverage}{2} = 1\ ns \tag{4.1}$$

4. Looking at the "traced" time (*TraceAverage*), we can see that no matter how many tracepoints we generate, the performance per tracepoint is not affected and remains steady.
5. The "traced" metrics show that the overhead of tracing synchronous OpenCL API calls is about 361 ns minus the base call time of 4.5 ns, which give us 356.5 ns for recording two UST tracepoints, or 178.2 ns per tracepoint.

4.5.3 Asynchronous Function Tracing Overhead

As displayed in Figure 4.2, recording asynchronous event timing metrics on OpenCL requires hooking a callback function to the event. The call to an enqueuing function will have the API function's enqueue start and end tracepoints recorded and, when the command is done processing by the GPU, another custom tracepoint that contains the timing metrics will be created. The overhead of such event is expected to be larger than any other synchronous API call since it contains a tracepoint with 44 bytes of payload. This payload has to be generated from 4 calls to *clGetEventProfilingInfo* and 2 calls to *clGetEventInfo*. This adds up to the overhead of tracing but it is important to keep in mind that the asynchronous calls are generally performed only a limited number of times per iteration in a computation as compared to synchronous calls. Table 4.3 shows the average time and standard deviation for various executions of data read from the device to the host using the *clEnqueueReadBuffer* function, configured for blocking read as a benchmark. Since we have previously shown in Table 4.2 that the performance of tracing is unaffected by the number of tracepoints to record, this table will show the performance for a fixed number of iterations with different buffer sizes in order to compare the overhead of tracing as a function of the size of the data transferred.

This approach shows that the weight of tracing such calls is constant as a function of the size of the buffer being transferred, and the tracing overhead impact per tracepoint is lowered

Table 4.3 Asynchronous OpenCL API function overhead benchmark

Buffer Size (Byte)	Base Ave. (ns/call)	Base Std Dev. (ns/call)	Preload Ave. (ns/call)	Preload Std Dev. (ns/call)	Trace Ave. (ns/call)	Trace Std Dev. (ns/call)	Preload Overhead (ns/call)	Trace Overhead (ns/call)
4	149.51	1.47	164.7	1.1	7000.6	261.8	15.2	6851.1
40	158.99	0.92	168.7	1.3	7026.8	289.5	9.7	6867.8
400	156.15	1.50	174.7	1.3	7269.3	240.6	18.5	7113.2
4×10^3	188.44	1.14	226.7	1.3	7043.6	244.2	38.3	6855.2
4×10^4	1499.76	5.47	1503.3	5.6	8393.0	227.2	3.6	6893.3
4×10^5	17805.67	134.31	17862.1	16.1	25404.7	276.3	56.4	7599.0

Sample size = 100, Loop size = 1000

when using larger buffers. Unfortunately, the OpenCL read command time is greatly increased and small buffer reads suffer the most in relative terms. On our test bench, with the `Beignet` OpenCL drivers, the time measurements are more accurate since the PCI-Express communication latency is not present for integrated graphics.

Measurements have shown that this overhead is caused by two different sources: the calls to the *clGetEventProfilingInfo* function for gathering the start and end times on the device (approximately 1600 ns per call) and the overhead of providing an event to the OpenCL enqueue function (again, twice the same 1600 ns `ioctl` syscall). The *Beignet* library performs a heavy *syscall* to access and write the recorded start and end time of asynchronous functions: this results in a cost of four times 1600 ns, more than 90% of the overall tracing overhead. The recording of the events with LTTng-UST has been measured to be insignificant, as shown in Table 4.2. Therefore, when tracing is disabled, the data gathering is not performed and the call the UST tracepoint is also not performed. Avoiding the callsoverwhelm to extract information when tracing is disabled keeps the CLUST library overhead as low as possible when not active.

This very high overhead for retrieving two long integers from the GPU may be reduced by modifying the driver and possibly the GPU microcode to automatically retrieve this information at the end of the execution of the asynchronous command, when OpenCL profiling is enabled.

4.5.4 Real OpenCL Program Tracing Overhead

Knowing the cost of tracing individual synchronous and asynchronous OpenCL calls does not represent the typical overhead of tracing an OpenCL program. For testing purposes, we used a synthetic benchmark that performs 14 synchronous OpenCL calls and 2 asynchronous calls per iteration. The program generates an image of variable width and height, from one

simple run of a very light OpenCL kernel.

We measured the performance for CLUST tracing alone and then the overhead of tracing the operating system kernel with LTTng, for a unified trace overhead measurement. Results are displayed in Table 4.4.

Table 4.4 Real application tracing timing

Width (pixel)	Height (pixel)	Base Ave. (ns/iter.)	Base Std Dev. (ns/iter.)	Preload Ave. (ns/iter.)	Preload Std Dev. (ns/iter.)	CLUST Ave. (ns/iter.)	CLUST Std Dev. (ns/iter.)	CLUST + LTTng Ave. (ns/iter.)	CLUST + LTTng Std Dev. (ns/iter.)
1	1	35198	2162	36399	1941	50890	2297	58838	3364
10	1	35183	1916	35702	821	51883	2315	58265	3135
10	10	36031	1936	36890	1941	50758	1931	59619	3531
10	100	37937	1868	39067	169	55820	2731	61108	2645
100	100	56770	2440	59709	2135	75073	2661	84746	2232
1000	100	250694	3371	251165	3268	268726	3388	280299	4534
1280	720	1951826	4104	1951965	4443	1976916	4528	1988445	4747
1920	1080	4466096	6345	4466777	5597	4491589	5636	4511394	5509

Sample size = 1000, Loop size = 100

CLUST tracing

The expected CLUST tracing overhead is at least $14\times \sim 359ns + 2\times \sim 7030ns =\sim 19086ns$ per iteration. The results are displayed in Table 4.5.

As we can see, the overhead of tracing has a significant impact on small sized computation tasks. However, when the program is working on bigger images, for more typical resolutions such as 720p and 1080p, the relative tracing overhead factor is considerably reduced as compared to the complexity of the calculation itself. GPGPU applications are usually more efficient on larger data sets anyway, again because of the fixed startup costs of sending the commands to the GPU. Figure 4.5 shows the relative overhead (R_O from Equation 4.2) of tracing with CLUST as a function of the workload.

$$R_O = \frac{TraceAverage - BaseAverage}{BaseAverage} \tag{4.2}$$

We can also verify the expected tracing overhead computed at the beginning of this section ($\sim 19086ns$ overhead per call); we can see that the measured tracing overhead for the real OpenCL programs varies from 14727ns to 25494ns and averages at 18990ns, a measurement that is within 0.5% of the expected tracing overhead.

Table 4.5 Overhead of using CLUST and CLUST + LTTng

Width (pixel)	Height (pixel)	Preload Overhead (ns/iter.)	CLUST Overhead (ns/iter.)	CLUST + LTTng Overhead (ns/iter.)
1	1	1201	15692	23640
10	1	520	16700	23082
10	10	859	14727	23588
10	100	1131	17883	23171
100	100	2939	18303	27976
1000	100	471	18032	29605
1280	720	139	25090	36619
1920	1080	681	25493	45298

Sample size = 1000, Loop size = 100

CLUST tracing + LTTng kernel tracing

The main advantage of our solution, as compared to existing CPU tracing and GPU tracing approaches, is the unified tracing capabilities. Our solution allows recording the CPU trace as well as the GPU trace in the same trace format. The gathered performance data is displayed in Table 4.4 under the *CLUST + LTTng* columns.

As Figure 4.5 shows, the tracing overhead of the unified CPU + GPU tracing is yet again very low for larger computations. One of the reasons why the unified tracing overhead does not significantly differ from the CLUST tracing overhead is because the GPU execution speed is not affected by CPU tracing. The overhead is therefore a bit higher than the CLUST tracing overhead alone but this difference fades as the workload size gets larger.

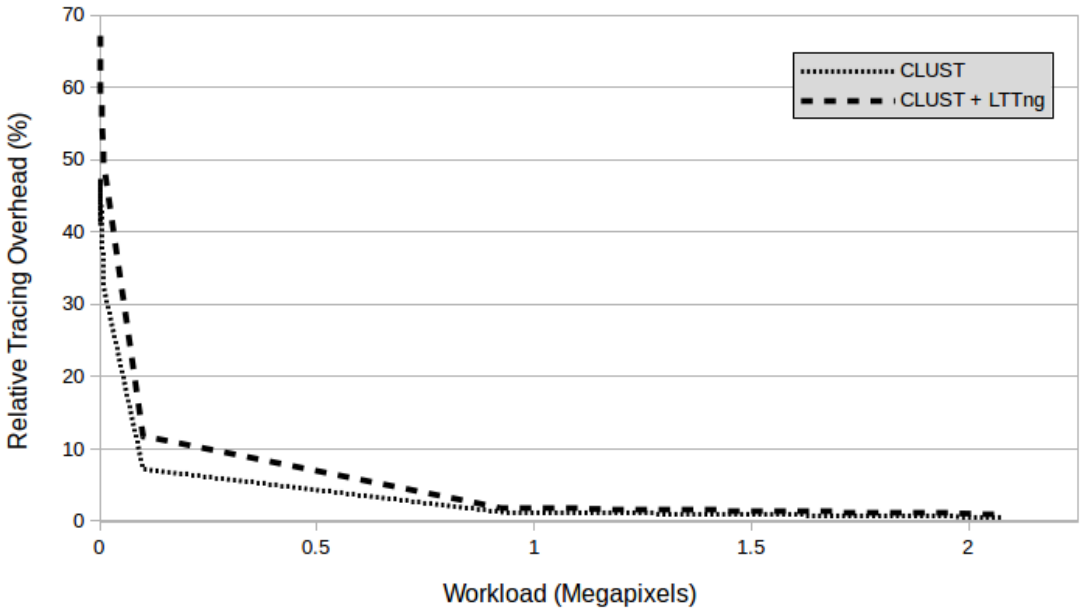
It is interesting to point out that the longer the iterations are, the more LTTng kernel events are recorded. This can be observed by the growth of the overhead per iteration in the *CLUST + LTTng Overhead* column of Table 4.5.

The addition of LTTng kernel tracing to the GPU trace allows programmers to find links between the system state and the GPU execution speed. This can lead to performance improvements and bug solving via trace analysis.

4.5.5 CLUST Use Cases

The biggest advantage of CLUST is not only its low overhead characteristics but also the new kind of information it provides to the programmers. Having access to a unified CPU-GPU trace and its analysis tools can help troubleshoot different types of problems or even help

Figure 4.5 Overhead of CLUST tracing and unified CLUST + LTTng tracing relative to the workload size



enhance application performance through GPU pipeline analysis.

Examples that depict this kind of troubleshooting scenarios and enhancements will be demonstrated in this section.

4.5.6 Typical Execution

As a reference for the following use cases, Figure 4.4 shows a close-up view of one iteration of the program that will be analyzed. It consists of a trace of the program that was presented in Section 4.5.4 and used to gather performance data. This iteration showed no abnormality and will therefore serve as a base for the upcoming use cases.

4.5.7 Abnormal Execution

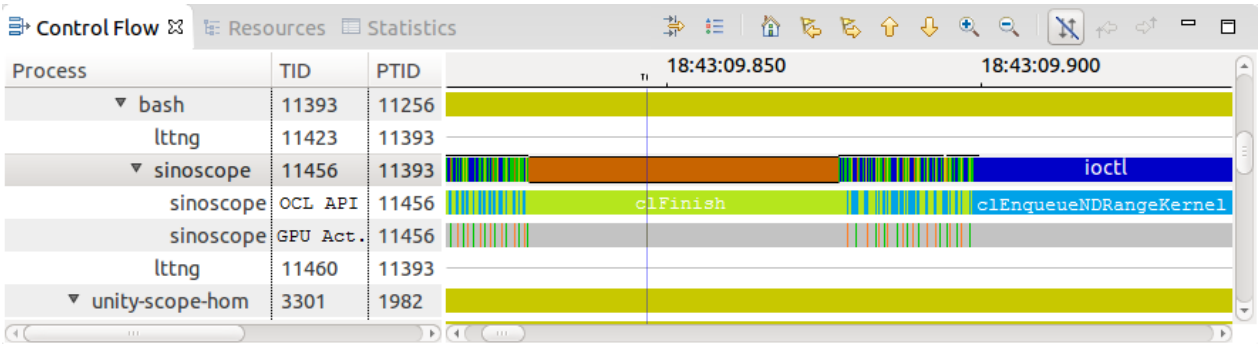
There are two primary types of abnormal executions. The first type occurs when the CPU is preempted, the second occurs when the GPU is being used by different processes.

CPU preemption

CPU preemption occurs when the scheduler gives CPU time to another process. For GPU intensive applications, this can lead to catastrophic performance because the CPU constantly

gets interrupted and put on wait for device (GPU in our case) when executing calculations or communicating with the device. Figure 4.6 shows an example of how a unified trace visualization tool would help spot this kind of latency. The orange section of the system kernel trace shows that the CPU is in the *WAIT_FOR_CPU* state while the call to a read buffer OpenCL function was completed. Also, the sections on the right and the left of the orange section show normal iteration times. A programmer could fix this problem by making sure that the thread that manages the OpenCL calls has a higher priority than normal, and making sure that no other process uses all other available resources.

Figure 4.6 Unified CPU-GPU trace view that depicts latency induced by CPU preemption



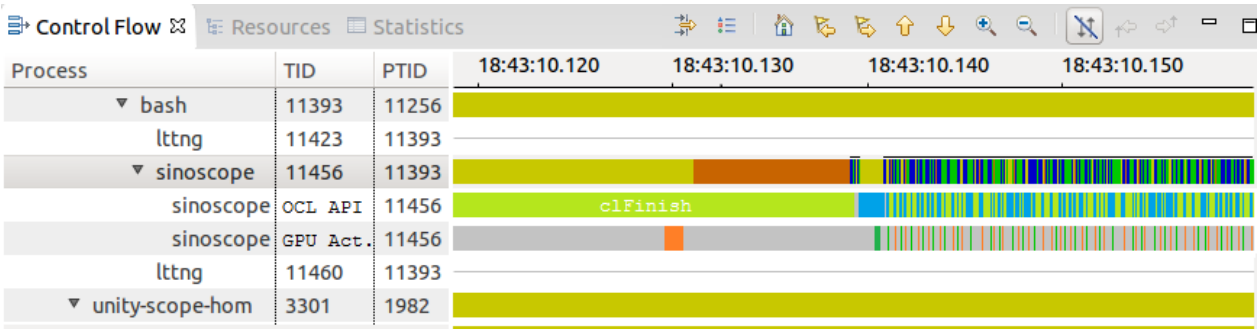
Shared GPU resources

A second issue that can be observed is when the GPU device is shared between multiple processes. The GPU's primary task is to deal with rendering the screen. This process can interfere with the OpenCL kernel execution, and data transfers between the host and the device. Figure 4.7 shows a great example and its effect on the system. In yellow, we can see that the system is in state *WAIT_BLOCKED* for a long time before being preempted (in orange). On the right side of Figure 4.7, we can see that the process is running normally. This can be caused by the execution of a concurrent application that uses the GPU resources. If the OpenCL program analyzed is critical, making sure that no other process interferes with it can fix this problem.

4.5.8 Non-Optimized Execution

Another kind of flaw that OpenCL analysis can outline it non-optimized OpenCL programs. Due to the different architectures of the GPUs, adapting to the device characteristics is a very important feature of OpenCL programs. Pipeline enhancement and kernel occupancy

Figure 4.7 Unified CPU-GPU trace view that depicts latency induced by GPU sharing



are just two of the main focuses for optimization. In both cases, the goal it to maximize the device and host resources utilization.

OpenCL pipeline enhancement

One of the best ways to optimize the speed of an OpenCL application is to properly use its pipeline. OpenCL has the capability to process simultaneously device communications and OpenCL kernel executions. Using OpenCL's asynchronous command enqueuing functions, it is possible to maximize the GPU resources usage. As we can see in both Figure 4.6 and Figure 4.7, the gray area of the *GPU Act.* represents idle time for the GPU. These views show that most of the computing and communication time is not being used.

Depending on the application type, the idle time can be reduced by dividing the calculations when possible and enqueuing multiple commands in different command queues to ensure that the device always has work to do. As an example, we divided the workload of our application benchmark and noticed a 28% speedup when running 13 command queues instead of one (with one `pthread` per command queue). We also measured a speedup of up to 72% for 12 command queues on a different dedicated GPU. The bigger speedup on the dedicated GPU can be explained by the larger overhead of interacting with the device: the latency to communicate with a dedicated GPU is higher than for an integrated graphics GPU. Therefore, dedicated GPU configurations are well suited for pipeline enhancement. After 12 and 13 command queues and threads, we started to notice a slow down when we reached the device saturation point.

CLUST records the required information to allow the analysis and visualization of this kind of behaviour.

OpenCL kernel occupancy

Unfortunately, CLUST cannot currently extract and record the required metrics on the device regarding kernel occupancy since this information is not available in the OpenCL standard. However, having a record of the *clEnqueueNDRangeKernel* parameters, and knowing the device architecture, can help evaluate how the computation was divided and if compute blocks were executed while using a maximum of available resources on the GPU.

4.6 Conclusion & Future Work

In this paper, we explored the GPU tracing challenges and addressed the need for a unified CPU-GPU tracing tool. This allows programmers to have a better global overview of the execution of OpenCL powered applications within the whole system. We demonstrated the very low overhead of intercepting OpenCL calls to insert LTTng-UST tracepoints, not only when tracing is inactive, but also when tracing is enabled. This minimizes the impact on the system and maintains the spirit of LTTng as a very low overhead tracing tool. To our knowledge, we presented the only unified GPU-CPU tracing tool available in the open literature.

In complement to this, we demonstrated the analysis and visualization capabilities that a unified trace can provide, and the type of problems that it can help uncover.

As this is a valid solution for OpenCL tracing, future enhancements would be to implement OpenGL and CUDA UST tracing in order to have a global understanding of all GPU activity. Kernel space driver tracing could also be a great improvement for understanding the host's queue. Trace Compass's unified views also need to be fine-tuned. This tracing model can serve not only for GPU tracing but also paves the road for other heterogeneous computing analysis.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous aborderons une discussion à propos de l'utilisation de CLUST et explorerons les contributions complémentaires apportées.

5.1 Cas d'utilisation

Dans l'article présenté dans le Chapitre 4, nous avons démontré le faible surcoût d'utilisation de CLUST pour enregistrer les appels à la librairie `OpenCL` ainsi que la facilité d'instrumentation de n'importe quelle application utilisant OpenCL. Le complément que représente l'ajout d'une trace CLUST à une trace noyau LTTng a aussi démontré comment il permettait de visualiser et de trouver des problèmes d'implémentation. D'autres cas d'utilisation peuvent alors être considérés.

5.1.1 Instrumentation du système en entier

Contrairement aux outils de traçage GPU disponible, il est possible avec CLUST de surcharger la variable système de chargement dynamique *LD_PRELOAD* afin de pouvoir enregistrer toutes les interactions entre le système et la librairie OpenCL. Les autres outils nécessitaient effectivement d'analyser un programme à la fois. De plus, ils devaient aussi être lancés avec l'outil d'analyse pour pouvoir être analysés. Avec CLUST, les programmes peuvent tous être instrumentés en tout temps et donc permettre l'analyse à n'importe quel moment.

5.1.2 Mode enregistreur de vol

LTTng permet des fonctionnalités comme l'enregistreur de vol. Cette fonctionnalité permet d'enregistrer constamment dans des tampons et d'enregistrer le contenu des tampons seulement lorsqu'un événement intéressant survient. Grâce au faible surcoût de traçage de la solution proposée (CLUST) combiné au faible surcoût de LTTng, cette technique peut très bien être appliquée à des systèmes critiques pour mieux comprendre le contexte qui a mené à un problème quelconque. La trace servira alors de boîte noire : contenant le maximum d'informations nécessaires à diagnostiquer les causes de l'erreur tout en évitant de remplir inutilement le disque dur d'une trace pouvant devenir énorme si l'enregistrement est fait constamment.

5.2 Autres contributions

En plus de l'article présenté dans le Chapitre 4, d'autres contributions significatives furent apportées, en voici la nature.

5.2.1 Trace CodeXL

Comme présenté dans la revue de littérature, le format d'enregistrement de la trace `CodeXL` (l'outil de traçage fourni par `AMD`) est en format texte. Cette caractéristique rend son interprétation assez facile. De plus, la synchronisation de temps entre la source de temps utilisé pour la trace `CodeXL` semble être la même que celle utilisée par `LTTng` : l'horloge monotonique. Nous avons implémenté un interpréteur de trace `CodeXL` qui permet de décoder la trace et de l'afficher dans `Trace Compass`. En combinant une trace `LTTng` noyau et cette trace d'accélérateur graphique dans une *"Experiment"* grâce à `Trace Compass`, nous pouvons obtenir un niveau semblable d'information. Toutefois, cette option n'offre pas la performance d'analyse et la facilité d'utilisation de `CLUST`. Le format de la trace `CodeXL` ainsi que son format texte représente effectivement un goulot d'étranglement qui pourrait ralentir l'exécution du programme analysé si l'écriture des événements dans le fichier de trace dépasse les capacités du système.

5.2.2 Projet Beignet

Le projet `Beignet` est un projet à code source libre qui permet l'utilisation des nouveaux processeurs `Intel` dotés d'une carte graphique intégré pour OpenCL sur GNU/Linux. Il s'agit d'une librairie au niveau utilisateur qui interagit avec les pilotes graphiques `i915` situés dans le noyau du système d'exploitation.

Pendant la prise de métriques de performances, il fut noté que ces dernières ne reflétaient pas le surcoût réel attendu alors que le traçage LTTng-UST avec CLUST était activé. Suite à une investigation, le problème fut trouvé dans la librairie `Beignet` qui omettait d'appeler une fonction de rappel quand l'événement déclencheur de cette dernière était déjà atteint. Un correctif qui requiert une ré-implémentation de la gestion des rappels de fonction au sein de la librairie fut écrit et soumis à l'équipe en charge du projet. Ce correctif est accepté et fait maintenant partie de la librairie.

CHAPITRE 6 CONCLUSION

Dans ce dernier chapitre, nous présentons un retour sur les travaux effectués, et la limitation de la solution proposée dans l'article du chapitre 4, pour finalement conclure avec des améliorations futures dont la solution pourrait profiter.

6.1 Synthèse des travaux

Les questions de recherche de ce mémoire avaient comme but de trouver des réponses aux deux principales variables inconnues identifiés au début de cet ouvrage : **Q.1** possibilité de tracer les systèmes hétérogènes CPU-GPU sans les ralentir significativement et **Q.2** possibilité de synchroniser les traces utilisant des différentes sources de temps.

Dans l'introduction, un objectif principal ainsi que des sous-objectifs sont déterminés afin de pouvoir répondre à ces questions de recherche. Ils peuvent être résumés comme suit : fournir un outil de traçage qui permet d'unifier les traces CPU et GPU en une unique trace qui permet une visualisation et analyse consolidées de ces deux éléments distincts, et ce en ayant un surcoût aussi faible que possible.

Pour répondre à cet objectif, nous avons commencé par présenter une revue de littérature approfondie sur le sujet du traçage en général ainsi que du traçage des accélérateurs graphiques. Cette revue de littérature nous permit alors de prendre connaissance des différents outils de traçage existants, non seulement pour le traçage CPU, mais aussi pour le traçage GPU. D'autres notions techniques furent alors abordées, notamment les différentes sources d'horloge pour l'estampillage temporel, l'architecture unique des accélérateurs graphiques ainsi que des techniques d'analyse et de visualisation de traces.

Puis, la méthodologie utilisée fut expliquée dans le chapitre 3. Il fut question de la configuration expérimentale utilisée pour l'obtention des métriques de performance ainsi que des méthodes d'acquisition de données qui furent utilisées. Une explication du chargement dynamique de librairie fut aussi fournie afin d'illustrer le fonctionnement de la solution proposée.

Ensuite, le chapitre suivant proposait un article contenant l'essentiel de la solution. Les avantages des plateformes de traçage ont étés explorés pour ensuite faire place aux choix d'architecture en fonction des avantages et désavantages des solutions disponibles. Cette section est suivie des détails d'implémentation de la solution, et pour finir les résultats expérimentaux furent présentés. Ces derniers nous démontrent le faible impact du traçage qu'offre la librairie CLUST. C'est dans cette section que le surcoût du traçage unifié est mesuré et qu'on peut

en constater le faible coût par rapport aux données récoltées. L'accomplissement de l'objectif principal de ce mémoire peut alors être confirmé.

Dans le dernier chapitre, une discussion générale aborde les options d'utilisation de la librairie CLUST en considérant son faible coût de traçage. Les contributions connexes à la contribution principale sont aussi présentées.

En plus des questions de recherche et des objectifs de départ, nous avons pus confirmer les hypothèses de départ :

- **H.1**, qui supposait la possibilité d'enregistrer les données des événements asynchrones.
- **H.2**, qui suggérait LTTng-UST comme outil d'enregistrement de traces pour ses caractéristiques de performance.
- **H.3**, qui supposait que les méthodes de synchronisation de traces réseau pouvaient s'appliquer à la situation du GPU où le périphérique possède une base de temps différente de l'hôte.

6.2 Limitations de la solution proposée

Bien que la solution proposée puisse enregistrer toutes les interactions avec la librairie OpenCL, elle ne permet malheureusement pas d'enregistrer les interactions des autres librairies telles qu'OpenGL et autres librairies au niveau utilisateur utilisant la carte graphique telles que la nouvelle spécification d'API graphique `Vulkan`. Aussi, le traçage d'applications utilisant CUDA n'est pas possible : il s'agit aussi d'une librairie qui peut émettre des commandes de calcul au GPU et qui pourrait interférer avec la performance d'applications OpenCL. Par contre, la technique utilisée et expliquée dans ce mémoire représente un bon modèle pour l'enregistrement des autres APIs.

Aucune technique d'analyse de la trace unifiée n'a été présentée. Nous nous sommes limités à présenter une méthode de visualisation de cette dernière. Bien que le simple fait de pouvoir visualiser une trace procure un avantage net, des algorithmes d'analyse peuvent aussi permettre d'établir des liens pertinents et permettre la compréhension approfondie de la trace.

La collecte de données de performance n'est pas effectuée dans la cette version de CLUST. Il est donc impossible pour l'utilisateur de connaître le pourcentage d'utilisation des cœurs de calcul et de la mémoire graphique de son programme. Ces informations peuvent aider à découvrir des problèmes majeurs comme une mauvaise segmentation des blocs de calcul et une mauvaise utilisation de la mémoire en cas de mémoire insuffisante.

6.3 Améliorations futures

Pour répondre aux limitations énumérées dans la section 6.2, deux options se présentent. La première serait d'utiliser les mêmes techniques et concepts utilisés pour la réalisation de la librairie de traçage proposée dans cet ouvrage et de les appliquer aux autres librairies qui utilisent les accélérateurs graphiques. De cette façon, un maximum d'information pourrait être accessible et facilement comparable. Ainsi, pour le traçage de CUDA, l'utilisation de leur API de profilage CUPTI pour instrumenter les programmes CUDA pourrait s'appeler "CUUST". OpenGL pourrait aussi être instrumenté grâce aux points de trace LTTng-UST et alors former la librairie "GLUST". La seconde option serait d'instrumenter plus en profondeur les pilotes de la carte graphique pour être en mesure d'obtenir des informations complémentaires qui permettraient d'obtenir plus d'informations que le suggère le tableau 4.1 pour les pilotes `i915` d'`Intel`. Idéalement, l'obtention de la même quantité d'information qu'avec CLUST, c'est-à-dire des métriques temporelles de l'exécution des commandes au sein même du GPU, pourrait combler tous les besoins de traçage GPU.

Comme mentionné par Juckeland (2012), l'association de données de compteurs de performance GPU à la trace pourrait aussi être un ajout significatif pour les programmeurs qui désirent améliorer la performance de leur application utilisant des accélérateurs graphiques. Malheureusement, aucun standard ouvert (comme OpenCL ou OpenGL) ne permet d'obtenir ces données de performance : il faudrait alors utiliser les APIs de profilage rendus disponibles par les manufacturiers de cartes graphiques pour obtenir de telles informations.

RÉFÉRENCES

(2013) TAU - Tuning and Analysis Utilities. Department of Computer and Information Science, University of Oregon. En ligne : https://www.cs.uoregon.edu/research/tau/home.php

(2012) Introducing Titan | The World's #1 Open Science Supercomputer. US Department of Energy : Office of Science. En ligne : https://www.olcf.ornl.gov/titan/

AMD, *White Paper | AMD Graphics Cores Next (GCN) architecture*, Advanced Micro Devices Inc., jun 2012. En ligne : http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

AMD, *AMD Accelerated Parallel Processing*, Advanced Micro Devices Inc., 2013. En ligne : http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

Instruments User Guide, Apple Inc., oct 2014. En ligne : https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/

R. Capriotti. (2013, jun) Internet Explorer 10 is the Most Energy Efficient Browser on Windows 8. Microsoft Inc. En ligne : http://blogs.windows.com/ie/2013/06/05/internet-explorer-10-is-the-most-energy-efficient-browser-on-windows-8/

W. Cohen. (2012) Product Overviews : Using SystemTap. Red Hat. En ligne : http://youtu.be/nRehkd22j78

M. Desnoyers, *Common Trace Format (CTF) Specification (v1.8.2)*, 1er éd., EfficiOS, 2014. En ligne : http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md

——, "Low-impact operating system tracing", Thèse de doctorat, École Polytechnique de Montréal, Déc. 2009.

M. Desnoyers et M. R. Dagenais, "Lockless multi-core high-throughput buffering scheme for kernel tracing", *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 3, pp. 65–81, Déc. 2012. DOI : 10.1145/2421648.2421659. En ligne : http://doi.acm.org/10.1145/2421648.2421659

——, "Synchronization for fast and reentrant operating system kernel tracing", vol. 40,

no. 12, pp. 1053–1072, 2010. En ligne : http://onlinelibrary.wiley.com/doi/10.1002/spe.991/abstract

R. Dietrich, F. Schmitt, R. Widera, et M. Bussmann, “Phase-based profiling in gpgpu kernels”, dans *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Sept 2012, pp. 414–423. DOI : 10.1109/ICPPW.2012.59

W. C. Don Domingo, *SystemTap 2.7 - SystemTap Beginners Guide : Introduction to SystemTap*, 2013.

N. Ezzati-Jivan et M. R. Dagenais, “A stateful approach to generate synthetic events from kernel traces”, *Advances in Software Engineering*, vol. 2012, pp. 1–12, 2012.

——, “Cube data model for multilevel statistics computation of live execution traces”, *Concurrency and Computation : Practice and Experience*, vol. 27, no. 5, pp. 1069–1091, 2015.

——, “A framework to compute statistics of system parameters from very large trace files”, *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 1, pp. 43–54, Jan. 2013. DOI : 10.1145/2433140.2433151. En ligne : http://doi.acm.org/10.1145/2433140.2433151

S. E. Fagan, “Tracing bsd system calls”, *Dr.Dobb’s Journal*, vol. 23, no. 3, p. 38, 03 1998, copyright - Copyright Miller Freeman Inc. Mar 1998 ; Last updated - 2014-05-22 ; CODEN - DDJOEB. En ligne : http://search.proquest.com/docview/202719549?accountid=40695

D. Goulet, *Unified kernel/user-space efficient Linux tracing architecture : David Goulet*, 2012.

B. Gregg. (2014, may) strace wow much syscall. En ligne : http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

——. (2014, nov) From DTrace to Linux, Brendan Gregg (Netflix), TracingSummit2014. En ligne : http://tracingsummit.org/w/images/9/9d/TracingSummit2014_FromDTraceToLinux2.pdf

B. Gregg et J. Mauro, *DTrace ; dynamic tracing in Oracle Solaris, Mac OS X, and FreeBSD*, 06 2011, vol. 26, no. 3, copyright - Copyright Book News, Inc. Jun 2011 ; Last updated - 2013-07-02. En ligne : http://search.proquest.com/docview/869984396?accountid=40695

K. Group, *OpenCL 1.0 C++ Header*, 1er éd., Khronos Group, 2010. En ligne : https://www.khronos.org/registry/cl/api/1.0/cl.h

M. Harris. (2014, feb) 5 things you should know about the new Maxwell GPU architecture. Nvidia Corp. En ligne : http://devblogs.nvidia.com/parallelforall/5-things-you-should-know-about-new-maxwell-gpu-architecture/

Intel Corp. (2015) Intel® VTune™ Amplifier XE : Getting started with OpenCL* performance analysis on Intel® HD Graphics. En ligne : https://software.intel.com/en-us/intel-vtune-amplifier-xe/details

M. Jabbarifar, "On line trace synchronization for large scale distributed systems", p. 148, 2013, copyright - Copyright ProQuest, UMI Dissertations Publishing 2013 ; Last updated - 2014-09-23 ; First page - n/a. En ligne : http://search.proquest.com/docview/1561560787?accountid=40695

G. Juckeland, "Trace-based performance analysis for hardware accelerators", dans *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, et M. M. Resch, éds. Springer Berlin Heidelberg, 2012, pp. 93–104.

"OpenCL - the open standard for parallel programming of heterogeneous systems", https://www.khronos.org/opencl/, Khronos Group, accessed : 2015-02-09.

Khronos Group Inc., *OpenCL Reference Pages*, Khronos Group Inc., 2011. En ligne : https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/

Kishonti Ltd. (2015) CompuBench - performance benchmark for various compute APIs (OpenCL, RenderScript). Kishonti Ltd. En ligne : https://compubench.com

M. Lavelle. (2012, oct) U.S. Lab's "Titan" Named World's Fastest Supercomputer. National Geographic News. En ligne : http://news.nationalgeographic.com/news/energy/2012/10/121029-titan-fastest-supercomputer/

P. Mistry, C. Gregg, N. Rubin, D. Kaeli, et K. Hazelwood, "Analyzing program flow within a many-kernel opencl application", dans *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, série GPGPU-4. New York, NY, USA : ACM, 2011, pp. 10 :1–10 :8. DOI : 10.1145/1964179.1964193. En ligne : http://doi.acm.org/10.1145/1964179.1964193

Z. Nanhai. (2015, jan) Beignet. En ligne : http://www.freedesktop.org/wiki/Software/Beignet/

NVIDIA, *CUPTI : : CUDA Toolkit Documentation*, 6e éd., NVIDIA, 2014. En ligne : http://docs.nvidia.com/cuda/cupti/index.html

NVIDIA Corp. (2015) PhysX FAQ | NVIDIA. En ligne : http://www.nvidia.com/object/physx_faq.html

J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, et J. Phillips, "Gpu computing", *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008. DOI : 10.1109/JPROC.2008.917757

A. Palnitkar, P. Saggurti, et S.-H. Kuang, "Finite state machine trace analysis program", dans *Verilog HDL Conference, 1994., International*, Mar 1994, pp. 52–57. DOI : 10.1109/IVC.1994.323748

B. Poirier, R. Roy, et M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events", *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, pp. 75–87, Août 2010. DOI : 10.1145/1842733.1842747. En ligne : http://doi.acm.org/10.1145/1842733.1842747

G. Price et M. Vachharajani, "Large program trace analysis and compression with zdds", dans *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, série CGO '10. New York, NY, USA : ACM, 2010, pp. 32–41. DOI : 10.1145/1772954.1772961. En ligne : http://doi.acm.org/10.1145/1772954.1772961

K. Roth, S. Patel, et J. Perkinson, "The impact of internet browsers on computer energy consumption", 2013. En ligne : http://dev.bowdenweb.com/ua/browsers/ie/m/docs/the-impact-of-internet-browsers-on-computer-energy-consumption/the-impact-of-internet-browsers-on-computer-energy-consumption-2013-06-fcs3s.pdf

G. Singer. (2013, mar) The history of the modern graphics processor. En ligne : http://www.techspot.com/article/650-history-of-the-gpu/

A. Spear, M. Levy, et M. Desnoyers, "Using tracing to solve the multicore system debug problem", *Computer*, vol. 45, no. 12, pp. 60–64, Dec 2012. DOI : 10.1109/MC.2012.191

J. Stenmark, "Program execution trace compression analysis tool", p. 100, 2010, copyright - Copyright ProQuest, UMI Dissertations Publishing 2010 ; Last updated - 2014-01-09 ; First

page - n/a. En ligne : http://search.proquest.com/docview/518928723?accountid=40695

strace project, *strace(1) Linux Manual Pages*, 2010. En ligne : http://man7.org/linux/man-pages/man1/strace.1.html

D. Toupin, "Using tracing to diagnose or monitor systems", *IEEE Software*, vol. 28, no. 1, pp. 87–91, Jan 2011, copyright - Copyright IEEE Computer Society Jan/Feb 2011 ; Last updated - 2011-07-26 ; CODEN - IESOEG. En ligne : http://search.proquest.com/docview/818397878?accountid=40695

A. Verge, "Tracage logiciel assiste par materiel", Mémoire de maîtrise, 2014.

H. Waly et B. Ktari, "A complete framework for kernel trace analysis", dans *Electrical and Computer Engineering (CCECE), 2011 24th Canadian Conference on*, May 2011, pp. 001 426–001 430. DOI : 10.1109/CCECE.2011.6030698

F. Wininger, "Conception flexible d'analyses issues d'une trace systeme", p. 85, 2014, copyright - Copyright ProQuest, UMI Dissertations Publishing 2014 ; Last updated - 2014-10-03 ; First page - n/a. En ligne : http://search.proquest.com/docview/1564022126?accountid=40695

ANNEXE A Exemple de trace CodeXL

```
1  TraceFileVersion=3.0
2  ProfilerVersion=3.0.2600
3  Application=/home/user/openclProgram
4  ApplicationArgs=
5  WorkingDirectory=/home/user
6  UserTimer=False
7  OS Version=Ubuntu 12.04.4 LTS \n \l
8  DisplayName=Mar-25-2014_13-40-22
9  ExcludedAPIs=
10 =====AMD CodeXL ocl API Trace Output=====
11 2427
12 43529
13 [...]
14 39 clSetKernelArg 127968390288 127968390356
15 39 clSetKernelArg 127968390589 127968390667
16 66 clEnqueueNDRangeKernel 127968399104 127968477647 4592
   CL_COMMAND_NDRANGE_KERNEL 127968416735 127968431626 127968782928
   127969537706 0 0xd98730 0 0x13928f0 Turks 0xda01f0 sinoscope_kernel
   {512,512} {NULL}
17 51 clFinish 127968493560 127969544649
18 52 clEnqueueReadBuffer 127969551023 127969928254 4595 CL_COMMAND_READ_BUFFER
   127969554150 127969557809 127969696465 127969923465 0 0xd98730 0 0x13928f0
   Turks 786432
19 =====AMD CodeXL ocl Timestamp Output=====
20 2427
21 43529
22 [...]
23 CL_SUCCESS = clSetKernelArg(0xda01f0;9;4;0x8c2360)
24 CL_SUCCESS = clSetKernelArg(0xda01f0;10;8;[0xd9af00])
25 CL_SUCCESS = clEnqueueNDRangeKernel(0xd98730;0xda01f0;2;NULL;[512,512];NULL;0;
   NULL;[0xd9e810])
26 CL_SUCCESS = clFinish(0xd98730)
27 CL_SUCCESS = clEnqueueReadBuffer(0xd98730;0xd9af00;CL_TRUE;0;786432;0
   x7f96a7525010;0;NULL;[0xd9eb70])
```